Formal Representation and Application of Software Design Information

Dissertation

Thomas M. Schorsch, B.S., M.S.,
Major, USAF
AFIT/DS/ENG/99-08

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/DS/ENG/99-08

Formal Representation and Application of Software Design Information

DISSERTATION

Presented to the Faculty of the Graduate School of Engineering
of the Air Force Institute of Technology
Air University in Partial Fulfillment of the
Requirements for the Degree of
Doctor of Philosophy

Thomas M. Schorsch, B.S., M.S.
Major, USAF

September, 1999

Approved for public release; distribution unlimited

AFIT/DS/ENG/99-08

Formal Representation and Application of Software Design Information

Thomas M. Schorsch, B.S., M.S.,
Major, USAF

Approved:

Thomas C. Hartrum 19 Aug 99
Thomas C. Hartrum, Chairman

Robert P. Graham Jr 19 Aug 99
Robert P. Graham, Jr.

Scott A. 19 Aug 99
Scott A. DeLoach

Aihua Wood 19 Aug 99
Aihua W. Wood

Kirk A Mathews 19 AUG 1999
Kirk A. Mathews, Dean's Representative

Accepted:

Robert A. Calico
Dean, Graduate School of Engineering

Acknowledgements

To Susie and Michael- Thanks for making this worthwhile.

To Dr. Hartrum and Dr. Graham- Thanks for guiding me through.

Thomas Michael Schorsch

Table of Contents

List of Figures

List of Tables

AFIT/DS/ENG/99-08

Abstract

This dissertation describes the development of methods for formally representing and applying design information that enables user determined software design decisions to be automatically and correctly applied to software requirements producing a software design.

Formal methods for developing software use mathematical frameworks to specify, develop and verify software systems, especially safety critical systems where error free software is a necessity. A transformation system is a formal method that refines an abstract requirement specification into a concrete implementation by successively adding design decisions in the form of precisely verified design information. Current algebraic representations of design information (specifications, morphisms, and interpretations) and methods for applying algebraic specification design information (diagram refinement) cannot correctly represent and apply higher-level design information.

This investigation develops innovative methods for constructing and refining structured algebraic requirement specifications, as opposed to individual specifications. A category of diagrams and diagram morphisms is developed and applied to algebraic specifications and morphisms that enables the structure of requirement specifications and design information to be dealt with explicitly. Parameterized diagrams enable large requirement specifications to be built using a parameter-passing analogy that is of a higher-level of abstraction than current methods. Diagram interpretations enable structured design information to be correctly represented and applied, including the refinement of parameterized diagrams and restructuring refinements.

The developed approach enables one to create a library of correctly represented software design information. Software could then be developed directly from the requirements by selecting design choices from the library and correctly applying the underlying design information. Such a transformation system would enable correct-by-construction software to be developed rapidly and easily.

Formal Representation and Application of Software Design Information

1 Introduction

Formal methods for developing software are based upon mathematical frameworks that enable software developers to specify, develop and verify software systems in a systematic rather than an ad hoc manner [Win90]. They can be used to specify requirements, designs and implementations precisely, as well as to detect ambiguities, incompleteness and inconsistencies within and between these specifications. They can further tie a requirement specification directly to a more concrete realization, thus ensuring the implemented program actually meets the requirements. A formally developed program may take longer to produce initially, but it should contain fewer errors and thus reduce the cost and time to validate.

An abstract view of formal software development, called the refinement or transformation approach [BCG83], is depicted in Figure 1-1. Informal requirements are used to develop a detailed initial formal specification that is then validated against the requirements. The initial requirement specification is abstract in that there are many equally valid implementations for the specification. Design information (architectural, data structure, algorithmic) is successively added to the formal specifications through a sequence of refinements that faithfully preserve the properties of the prior specifications. Finally, when enough design information has been added, the final formal specification can be encoded in the programming language of the developer's choice, then compiled and executed. The final specification (and its encoding) is a correct by construction realization of the initial requirement specification.

In a sense, the development of a refinement system is a logical progression from the development of compilation systems. Compilation systems and refinement systems both transform an "abstract specification" to a more concrete one. Design information in a compilation system provides the knowledge of how to use the instruction set of a computer and its data registers to implement the various programming language features such as control structures, procedure calls and parameter passing. A compilation system encodes that design knowledge implicitly and applies it automatically. Although improvements are constantly being made, it is evident that compilation systems encode and apply design information in a manner that is good enough for most programming efforts.

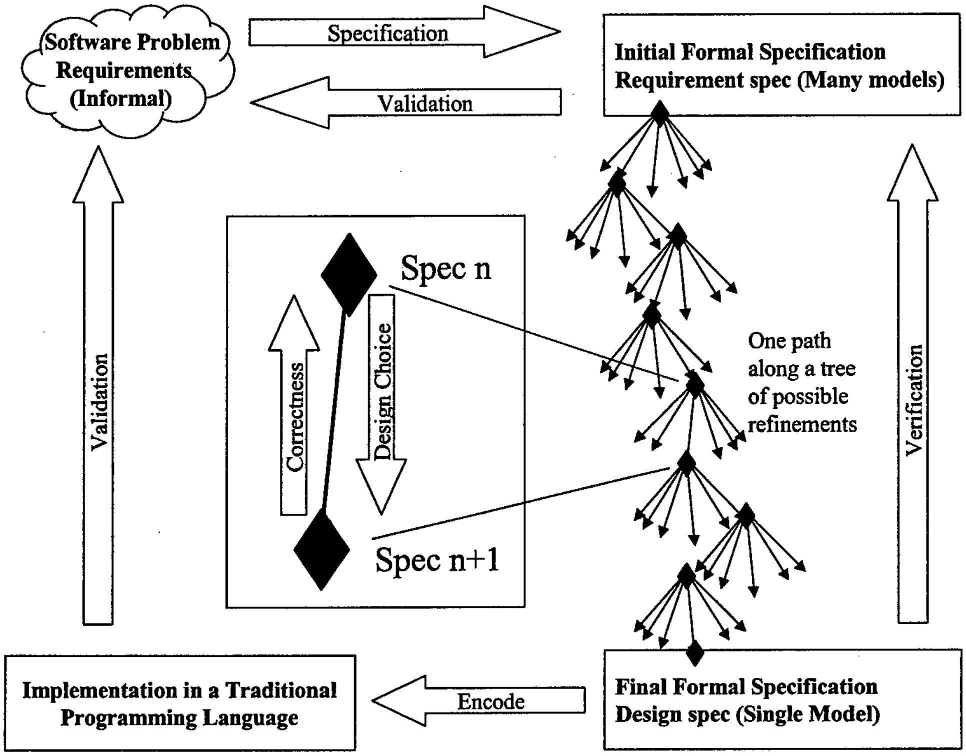

Software Problem Requirements (Informal)
Specification
Validation
Initial Formal Specification Requirement spec (Many models)
Validation
Spec n
Correctness
Design Choice
Spec n+1
One path along a tree of possible refinements
Verification
Implementation in a Traditional Programming Language
Encode
Final Formal Specification Design spec (Single Model)

Figure 1-1. Formal software development – The refinement paradigm

Unfortunately, it may not be practical (or possible) to hardwire specific architecture, data structure and algorithm design choices into a refinement system in a similar fashion unless the domain of applications is severely restricted. Instead, it may be possible to encode a variety of design information in a library so that a software developer can consciously choose what designs to apply to a specific requirement specification. Although the ultimate goal of a fully automated, general-purpose refinement system may never be realized, this dissertation aims to help close the gap between compilation systems and refinement systems, as formalization must precede automation.

1.1 Background

A formal software development system should be able to represent and apply design information formally and not just be able to represent and relate a requirement specification to a design specification in a formal manner. Such a development system could free software developers from having to develop and encode designs from first principles, as design decisions could be selected from the library and automatically applied in a manner that ensures their correctness. Without such a library, all design decisions that transform a requirement specification to a design specification must be developed and applied from scratch. (Each refinement step must be manually developed and individually verified.) With such a library, the design information is encoded and verified once and then can be used repeatedly in a manner that is guaranteed to preserve correctness.

In a formal transformation system, a formal representation is needed for the requirement specification, the design information, and the result of applying the design information to the requirement specification. These formal representations and the application of design information to requirement specifications must be consistent with each other. This dissertation uses *algebraic specifications* and *specification morphisms* [EMCO92, EMO92, Wir90] for representing requirements and design information. An algebraic specification consists of a set of sorts (abstract data types), a set of operations (abstract functions), and a set of axioms (logical expressions over the sorts and operations). A specification morphism maps the *signature* elements of a source specification (the sorts and operations) to those in a target specification such that the translated axioms of the source are theorems (logical consequences) of the axioms of the target.

1.2 Limitations of Current Techniques

While the techniques for developing refinements (implementations) of simple algebraic specifications are well known [EhR82, San88, Wir90, and BKL91], these techniques do not scale well for larger specifications. Specifically, structured (aggregate) specifications, which are formed by combining many smaller component specifications, are difficult to refine because of the sheer number of sorts, operations, and axioms in the aggregate specification. This section describes in general terms some of the problems that current approaches have in developing and refining structured specifications that the research presented in this dissertation addresses.

Constructing structured specifications

While the requirements of a small problem can easily be specified using a single algebraic specification, the more complex requirements of a larger problem may necessitate developing an aggregate specification from a collection of smaller component specifications. The current approaches for developing aggregate specifications are too low-level for developing large aggregates or are based on parameterized specifications whose instantiation does not retain structure.

The low-level approach for developing aggregate specifications essentially lists the component specifications and their relationships with each other. Each component specification is a node and each relationship is an arc and the collection of node and arcs form a *diagram* that represents the requirement structure. The diagram of components can then be used to form the aggregate specification by collapsing all of that structure into a single specification that essentially contains all of the sorts, operations and axioms of the component specifications.

An example of a diagram consisting of 20 nodes (specifications) and 20 arcs (specification morphisms) is depicted in Figure 1-2. This represents the data structure of a Petri Net [Pet77] consisting of a set of places, a set of transitions, a bag of input arcs from places to transitions, a bag of output arcs from transitions to places, and a map from places to natural numbers representing the initial marking of the Petri Net. Creating such a diagram structure by listing its arcs and nodes is too low-level, error prone, and non-intuitive.

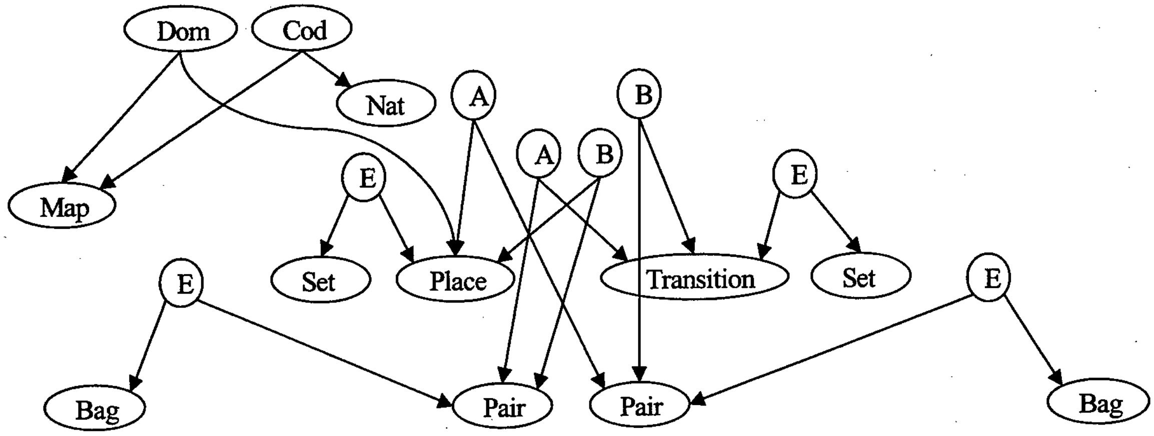


Figure 1-2. Example diagram: Petri Net data structure

The parameterized specification approach for developing aggregate specifications, on the other hand, iteratively builds up the final specification by extending a specification with additional signature elements and properties. In this approach a higher level parameter passing syntax is used to instantiate parameterized specifications in order to create the aggregate specification. The overall structure is never explicitly realized because the process involves adding to an existing specification rather then creating the structure of components. Thus in the parameterized specification approach, while it may be easier to develop the aggregate specification, the structure is lost by the construction method.

Refining structured specifications

Once a structured requirement specification has been created, the problem switches to refining the structured specification. This can be accomplished by refining the component specifications in a compatible way and then combining the individual refinements of the components into an aggregate refinement of the aggregate specification.

Unfortunately such an approach, called *diagram refinement* [SJ95], does not adequately take advantage of the structure of the specification as at heart only single specifications are being refined. Structure is used only to aggregate the individual specification refinements to form a refinement of the aggregate specification. Thus, for example, several specifications in the diagram in Figure 1-2 could not be refined as a unit using diagram refinement. This limits the type of design choices that can be made based on the initial requirement specification. While any individual specification may be free of implementation bias, the structure of a requirement specification contains a form of implementation bias when the diagram refinement approach is used.

1.3 Problem

This investigation is concerned with making it easier to develop and refine a structured requirement specification using algebraic specification methods. The goal of the research is to develop methods for formal representation and application of high-level design information. The structure of aggregate specifications is to be retained during development so that it can be used to guide refinements. Design information is also structured and (after being applied to a requirement specification) that structure is retained in the requirement specification. This enables refinements to substantially revise the structure of the requirements as well as simply adding design details.

1.4 Approach

The goal of the dissertation research was to develop methods that enable higher-level design information to be formally represented and applied. The approach taken is to consider *diagrams* of specifications and morphisms as the primary means of representing requirement information and design information. A diagram can represent both the content and the structure of the requirement and design information.

In order to accomplish the goal, a general theory of diagrams (independent of specifications and specification morphisms) is developed using category theory, which enables diagrams of nodes and arcs to be related to each other (via diagram morphisms) in a way that preserves both the structure and the content of the source diagram in the target diagram. The theory enables diagrams of specifications to be parameterized and instantiated where the result is a diagram, not a single specification. Thus it is the diagrams of specifications and specification morphisms that are created, extended, parameterized, and instantiated, not individual specifications. This approach is similar to the low-level diagram construction methodology in that the result of extension and instantiation is a diagram and not a specification. It is also similar to the parameterized specification approach in that a higher-level parameter passing syntax can be used to construct the diagram. This general diagram theory, when applied to specifications, forms the basis of the specification construction and refinement methodology developed in this dissertation.

Refinement is based on a diagram of specifications and morphisms instead of individual specifications. This added structure enables the refinement mechanism to take advantage of existing structure in a given requirement specification and enables restructuring refinements where the structure of the design specification differs appreciably from the structure of the requirement specification.

The design information is itself represented by a diagram of specifications and morphisms. A relationship between the design information and the requirement specification is established that maps both the structure and the content of the design information to the requirement specification. This relationship is then used to merge the design information with the requirement specification in order to create a more concrete design specification.

1.5 Limitations

The design information addressed in this dissertation is of the data structure variety only. Although the techniques can also be used to apply algorithm information, the representation and application of such design information are not addressed. Additionally, while the syntax and semantics are defined for some of the language constructs developed in this dissertation, an automated tool has not been built. Finally, this dissertation also does not address the transformation of a design specification to code.

1.6 Contributions

Contributions of this research include the following:

- A general theory of diagrams and the structure- and content-preserving diagram morphisms between diagrams based on category theory (Chapter 3).
- A collection of higher-level operations for creating, composing, relating and manipulating diagrams and diagram morphisms (Chapter 3).
- A category of diagrams of specifications and morphisms that enables diagrams of specifications and morphisms to be treated as if they were simple specifications and which can be used to apply the semantics of specifications and specification morphisms to diagrams of specifications and specification diagram morphisms (Chapter 4).
- A theory of parameterization for diagrams that subsumes all current (category theory based) specification parameterization mechanisms while providing new capabilities (Section 3.5, Section 4.2, and Chapter 5).
- A higher-level syntax for constructing diagrams based on parameterized diagrams and their instantiation (Chapter 5).
- A method for representing structured design information that enables the preconditions for applying the design information to be stated accurately (Section 4.3 and Chapter 6).
- A higher-level refinement mechanism based on structured design information that enables the design information to be automatically applied in a correct manner, given that the preconditions are met (Chapter 6).

1.7 Overview of Document

The body of the dissertation assumes that the reader is familiar with the terminology and notation of algebraic specification and refinement, models and category theory that are used throughout the document. Appendix A (Category Theory), Appendix B (Specifications and Models) and Appendix C

(Specification Development and Refinement) are included as a service to the reader who is unfamiliar with the basic terms and notations of category theory, algebraic specification, or specification refinement.

The remainder of this document is organized as follows:

- Chapter 2 provides some background material and describes the problems that current methods have with constructing and refining structured specifications.
- Chapter 3 defines a general theory of diagrams and diagram morphisms that underlies the later chapters. This chapter also defines the colimit operation over categories as well as the abstract notion of a parameterized diagram.
- Chapter 4 applies the general diagram theory developed in Chapter 3 to the category of specifications (*Spec*) and then extends the theory based on underlying category *Spec*. Specifically, the notion of parameterized (specification) diagrams and diagram interpretations are developed.
- Chapter 5 builds on the theory of parameterized diagrams by defining the syntax and semantics of a diagram statement that enables diagrams of specifications and morphisms to created, extended, parameterized and instantiated. This section also demonstrates that parameterized diagrams subsume the other forms of specification parameterization and instantiation presented in Section 2.3.
- Chapter 6 defines a new mechanism for representing and applying structured design information.
- Chapter 7 provides a summary and evaluation of the work done and highlights the contributions. It also gives some suggestions for future work in this area.
- Appendix D provides an example of the diagram construction method being used to create a large diagram. This example illustrates that the diagram construction method does not scale well.
- Appendix E describes existing parameterized specification research in greater detail than is done in the body of the dissertation (in Section 2.3) and compares it with the parameterized diagram developed in Section 4.2.
- Appendix F provides the code for the example diagram interpretations presented in Chapter 6.

2 Background

This chapter describes other research related to representing and applying design information. The latter sections describe the specific problems with creating and refining large structured requirement specifications that other approaches have and that this dissertation solves.

This chapter is structured as follows. Section 2.1 analyzes different formal methods representing how design information is currently represented and applied. Section 2.2 informally defines the terminology used in this dissertation and provides pointers to the more formal definitions of these terms in the appendices. Section 2.3 describes research on parameterized specifications. Section 2.4 describes research on developing large structured specifications. Section 2.5 describes research on refining large structured diagrams.

2.1 Related Work

In order to accomplish the goals of this dissertation, different formal methods were analyzed to discover how design information is represented and applied. This section presents an informal survey of some of the more prominent formal software development methods and analyzes them based on the design information that can be explicitly expressed and applied in the methodology. Section 2.1.1 describes the design information issues that form the basis of the analysis, Sections 2.1.2 through 2.1.5 present the findings based on an informal categorization of existing formal development methods, and Section 2.1.6 summarizes the findings.

2.1.1 Design information issues

There are several classic knowledge engineering issues that all knowledge-based systems have. The basic issues associated with design information in a formal transformation system are similar. These issues are listed below and form the basis of the analysis of the different formal development methods.

- What is the design information?
- How is it represented?
- How is it applied to solve a problem?

- How is it acquired or updated?

There are many different types of software design information. Software architectures are very high-level design information while local peephole-like program optimizations are very low-level. Between them are high-level abstract data type representation, high-level algorithm design representation, and statement or expression selection.

In many cases the actual higher-level design information is only represented implicitly in a set of lower-level design decisions. For example, constructing an algorithm is done implicitly when explicitly selecting a sequence of programming language statements. The concept of the algorithm does not exist except as an emergent property of the combined statement selections. The algorithm has no independent representation and it must be re-derived every time it is "applied".

Additionally, the design information that makes up a refinement could originate in a rule-based knowledge base, a catalog of design theories, be preprogrammed into the system as a transformation, or come from the designer's previous experience (no explicit representation). A refinement system may be extensible by the users of the system (it is obviously extensible by the developers of the refinement system itself) so that application developers can add their own design information, or the refinement system may be a "take it or leave it" black box. A design choice that transforms the requirement specification could range from the trivial, or non-intuitive (an "unfold" step) to the important and critical (selection of global search vs. generate and test). A particular design decision could be applied as a single refinement, or it could take several smaller refinements to implement a design decision, none of which are particularly helpful independently. The proof that design information can be (or has been) applied in a correct manner may range from very formal, to rigorous, to informal. Finally, the selection of a particular design decision could be specified by the designer in a declarative fashion, selected by the designer from a list of available choices, pulled from a designer's experience, or automated by the refinement system by blindly searching or intelligently covering the design space.

The next several sections describe some of the existing approaches to the refinement paradigm with a special emphasis on the design information that underlies a refinement step in the methodology.

2.1.2 Manual refinement, or posit-prove

In manual refinement, a design specification is developed by hand and then proved to be a refinement of the requirement specification that preserves the desired properties. Depending on the relative differences between the two specifications, the proof may be simple or extremely involved. Software development methodologies that use the posit-prove approach typically use one or more generic proof tools (associated with the specification language used by the methodology) that can be used to help the designer prove the relationship between the two specifications. Unfortunately, automated proof tools are often not up to the task for these systems [San88]. Automated proof checkers can be used to verify manually developed proofs, but they are also not as general as one would like.

The appeal of the posit-prove approach is that it is very general. The software developer has full control over the design information being added to the requirement specification (subject to the limitations imposed by the problem being solved, the representation capabilities of the specification language, and the complexity of the proof). Defining a process for developing refinements can scope the general "anything goes" approach down and provide software developers with a framework to guide their efforts. The following simplified steps are used in a typical VDM (Vienna Development Method) refinement [San88], which is an example of a posit-prove approach:

- Develop a specification SP' that is believed to be a refinement of specification SP.
- Define a retrieve function, *retr*, that maps data values in SP' to those in SP, one for each data type.
- Prove that *retr* is a total function and surjective (that every concrete data value has an abstract value and every abstract value has a concrete representation).
- Identify an operation f' in SP' for each operation f in SP.
- Prove that $\mathit{pre\text{-}f}(\mathit{retr}(v')) \Rightarrow \mathit{pre\text{-}f'}(v')$ for all concrete values of v', and all f, f' pairs. (Ensures the pre-conditions of each concrete operation, f', are equally or less restrictive than the preconditions of the associated abstract operation, f).

  Prove that $\mathit{pre\text{-}f}(\mathit{retr}(v'))$ AND $\mathit{post\text{-}f'}(v', w') \Rightarrow \mathit{post\text{-}f}(\mathit{retr}(v'), \mathit{retr}(w'))$), where $w' = f'(v')$, for all concrete values of v' and w', and all f, f' pairs. (Ensures the results, w', produced by each concrete operation, f', imply those produced by the associated abstract operations, f).

This well-defined approach, while tedious, has been used for a number of practical problems in academic settings as well as in commercial use [CW96]. Several specification languages and research efforts use refinement steps similar to those in the above list. The most well known are Z [Spi89], VDM [Jon90], and RAISE [NHWG89]. RAISE (Rigorous Approach to Industrial Software Engineering) is a second-generation formal development system built on the foundations of VDM. Z, the VDM specification language and the RAISE specification language are all model-based specification languages with implicit or explicit notions of pre- and post-conditions. VDM and RAISE specification languages use functions, relations and sets to build models of the system. Z models are based on set theory and use an interesting schema calculus to combine and relate specifications. Hayes [Hay92] compares VDM and Z refinement approaches in detail.

The posit-prove method does not encode any design information except that which is implicit in the manually developed specifications. All design decisions come from the experience of the developer, and their realization in a specification is achieved manually. Design knowledge cannot be reused or extended in the manual approach. This approach can be applied to any level of programming problem.

2.1.3 Transformation synthesis

In a transformation system, a transformation takes as input a specification, SP (or part of a specification), and returns a second specification, SP', with the appropriate design information added. A transformation is typically a syntactic manipulation of the specification. Individual transformations can be applied automatically using some set of rules or built-in knowledge base, or a designer can selectively choose which transforms to apply depending on which design choices are made. Each transformation comes with an associated proof obligation that ensures that the transformation can be (or has been) applied correctly. Many different refinement systems use the transformation synthesis approach. These systems have been loosely categorized in this dissertation into the following four basic approaches: catalog rule-based, encoded rule-based, generative rule-based and refinement calculus-based.

A catalog rule-based refinement approach encodes the design knowledge in a library of rules that can be applied to a specification SP in order to transform it into SP'. The transformation rules often have the following form: *If (a part of) SP matches pattern P1 then modify (a part of) SP so that it matches pattern P2*. Catalog rules can often be extended and modified. Like all knowledge bases, it is difficult to come up with a complete and consistent set of rules. Catalog rule-based transformation systems are often applied automatically, and the design space is restricted to the available rules and their mode of application. (In automated systems, some rules and design spaces may never be selected based upon other rules always being selected first). Catalog rule-based systems, despite their syntactic nature, tend to be less formal and more heuristically oriented than other approaches. It can be difficult to determine precise proof requirements for more complex, design rich rules.

In an encoded rule-based approach, the design knowledge is encoded as an algorithmic procedure that can be applied to SP to obtain SP'. These encoded rules can be more powerful than catalog-based rules as they can perform transformations that are more global in nature. In one sense, the encoded rule-based approach subsumes the catalog rule-based approach as any rule that can be represented and applied in an open and extensible library catalog system could be encoded and applied using a closed, procedural method (encoded rule-based approach). The encoded rule-based approach is also often less formal for the same reasons as the catalog rule-based approach. Encoded rules can be applied automatically or manually. Users of the transformation system cannot extend the encoded rules however. AFIT tool [HB94], and CIP [Par90] are examples of encoded rule systems.

In a generative rule-based system, there are only a few simple, general rules that cover all possible transformations of interest to the refinement system. Burstall and Darlington [BD77] describe a set of seven simple rules (fold, unfold, introduce a function or variable, rewrite expression, etc.) that, when applied in various combinations, can transform simple algorithms into more complex and efficient ones. Later work on a generative rule set that does a better job of preserving equivalence between transformations than does the fold/unfold rule was accomplished by Sherlis [Sch81]. The rules can be applied manually, via a (mostly) blind search, or via an informal design tactic. The generative rule-based approach is a special case of the encoded transformation approach where the number of transformations is severely restricted and each transformation is formally and precisely defined in terms of its effects and its proof obligations. Because of this, design information cannot be represented and applied by a single refinement step but is instead emergent from a sequence of refinement steps involving the smaller transforms.

The refinement calculus is an extension of the Guarded Command Language [Dij76], in which a pre-order relation is established for some program fragments that indicates which program fragments can be replaced by others. The refinement calculus was developed independently by several different researchers [Bac88, Mor90]. While this transformation approach does have a syntactic component, the preorder relationship is defined in terms of weakest precondition. Thus, the rules of the refinement calculus express transformations in which a program fragment can be substituted for another when the weakest precondition of one implies the weakest precondition of the other. Various laws in the refinement calculus are: strengthen post condition, weaken precondition, introduce assignment (or sequential composition, conditional, loop, or local block), and introduce or eliminate logical constants. Typically a software synthesis task in the refinement calculus starts with a single specification statement, contains many intermediate steps in which the "specification" consist of mixed specification statements and guarded command code, and ends up with a "specification" containing only guarded command statements. The laws serve to manipulate the specification statements (modifying, adding and removing them) and to introduce guarded command statements. As the refinement proceeds by using and applying the various laws, the individual design steps introduce individual guarded command statements that collectively ensure the post condition given input satisfying the weakest precondition. The rules can be applied manually, via a (mostly) blind search, or via an informal design tactic. As in the generative rule-based approach, the individual laws in the refinement calculus approach are formally defined and have precise proof obligations related to their use. The design information in the refinement calculus approach is of a higher level than the generative rules, but they are still on the level of individual statements, not algorithms or data structures.

Transformation systems, including examples of many of the above approaches, were surveyed by Partsch and Steinbrüggen [PS83] in 1983. A comparative study of different algorithm synthesis methods [SA89] surveyed 22 different published algorithm derivations spread over 7 different problems (insertion sort, N-queens, graph marking, convex hull, etc) using transformation approaches. In that study only about 30% of the published derivations were partially or completely developed by implemented transformation systems.

The transformation synthesis approach does not encode high-level design information very well, and only some of the approaches encode low-level design information explicitly. In the generative and refinement calculus approaches, the refinement rules reflect program transformations that rarely reflect an individual design decision other than the use of a particular statement. A series of transformations in the generative and refinement calculus approaches do reflect the design decisions, but that higher-level information is implicit and difficult to represent and apply mechanically. Thus, the implementation of a design decision using these two approaches cannot be reused in successive development efforts, but must be re-proved each time. Because of this limitation, these two approaches are used in a "search" fashion that is only effective for small problems or used to make specific spot improvements to algorithms and designs. Feather [Fea82] mitigated this problem somewhat with his research that used a pattern of the overall form of the desired result to trim the search space when the generative rules were applied automatically.

The catalog and encoded-rule approaches to transformation synthesis can encode limited higher-level design information. The more complex the design information, though, the more complex the rule and the fewer cases to which it can be applied. Often in order to use a complex rule a specification must first be "jittered" into a particular form. The syntactic form of the rules limits the catalog approach to design information that can be applied using pattern matching. The encoded approach can in theory encode any type of design information; it is limited in that it cannot be extended easily and that it may be difficult to jitter a specification into the proper form. Because of the limited abilities in terms of encoding design information, the transformation approach is useful mostly for refining existing designs or for developing lower-level implementations and specific algorithms. The developer must know which set of transformations to apply to accomplish a particular design decision, or the system can blindly search for an appropriate set of transformations, or systematically search using a desired goal or a design tactic.

2.1.4 Deductive synthesis

In the deductive synthesis approach, a program and its proof are developed in parallel, with the proof prescribing the program. First, a specification is translated into a theorem that contains a relationship stating the existence of the output variables given the input variables. Next, the theorem is proved in a constructive fashion. Finally, a program is extracted from the proof of the theorem.

When the existence proof of a program output is done in a constructive fashion, there exists an associated computational method for finding that output. In fact, as the theorem is being proved, the program is being built up. The proof techniques that are being used can be directly associated with programming constructs. For example, when the proof uses case analysis, mathematical induction and lemmas, the program uses conditionals (case, if-then-else), recursion (iteration) and procedures, respectively. As an example, a specification for sorting a list, sort(L), has "TRUE" as a precondition and "Permutation(L,s) $\wedge$ ordered(s)" as a post condition. A theorem of that specification is "$\forall$ L, $\exists$ s | permutation(L,s) $\wedge$ ordered(s)". Proving constructively that for all lists L, there exists a list s that is an ordered permutation of list L will also provide a computational method for finding such an ordered list. Different choices in the use and order of proof techniques result in different algorithmic implementations. The resultant program is guaranteed to be correct because the program and the proof of its correctness are developed in concert.

Examples of deductive proof systems include the Deductive Tableau Framework [MW92] and Nuprl [Con86]. The Deductive Tableau Framework for program synthesis has been implemented for at least three different logics and theorem provers [Wal97]. The Nuprl system for deductive synthesis is based upon a constructive logic developed by Martin-Lïf [Mar82]. In both cases, the resultant program is typical of automatically generated program code in that it must be refined and optimized further to remove simplistic implementation choices.

The deductive synthesis refinement approach encodes design knowledge indirectly as proof techniques. By choosing a particular proof technique, the theorem prover is implicitly choosing a design refinement in the eventual program. These design refinements are probably unlike the design refinements of the other approaches. The deductive synthesis approach appears similar in power to the generative rule approach and refinement calculus approach described above. Like these other methods, the design information implicit in the deductive synthesis approach must be re-proven each time. Because there are no proof techniques related to higher-level design decisions, this approach is limited to smaller problems where the proof development is manageable.

2.1.5 Classification synthesis

When performing classification synthesis, a specification is related to a design theory in a way that classifies it as an instance of the problem solved by the design theory. If a specification has certain properties identified by the design theory, then the specification and design theory can be combined, which adds design information to the specification. In some cases the design theory may have to be extended in order to match up with the application specification. In other cases the requirement specification must be extended in order to meet the needs of the design theory. When refining data types, for example, sets can be mapped to lists in a way that duplicates the properties of the set abstract data type using a list abstract data type. That is to say a Set ADT is interpreted as a List ADT that has been extended with additional set-like operations and behavior in order to implement the Set ADT. Any requirement specification that has a set in it can be combined with the set-as-list design information to indicate that the abstract Set of values is to be implemented by a list data structure. Other design decisions for implementing a set could be as an array, as a bit vector, as a tree, as a recognizer function, etc. If the implementation programming language has lists as an intrinsic data type, as does the programming language Lisp, then this aspect of the requirement specification need not be refined further. However, if the implementation language has pointers (reference types) and record types then a list ADT can be further refined as a linked-list. These design choices could be combined prior to their application.

When a design choice involves an algorithm, an operation in the requirement specification is mapped to some algorithm scheme (global search, divide and conquer, etc) that can be used to implement the operation. The mapping associates input and output types of the requirement specification operation to types in the algorithm theory. The algorithm scheme may require operations such as destructors and constructors for the data types to be mapped as well. These must be supplied before the algorithm theory can be combined with the operation in the requirement specification. Combining the algorithm theory with the specification operation constitutes a refinement step and design decision that implements the operation using the specified algorithm.

When accomplished by a refinement system, the design schemes may be hard-coded into the system or may be part of a knowledge base. An automated synthesis system may have preprogrammed into it the necessary steps to instantiate the parameterized design scheme. This parameter information can be selected from a predetermined list provided by the system or from a list specifically constructed based on the mapping from the problem to the algorithm scheme, or previously entered parameter information. Sometimes the needed parameter can be constructed automatically based on previous information. Examples of classification synthesis include the Data Type REfinement system (DTRE) [BG91], the Kestrel Interactive Development System (KIDS) [Smi90a, Smi90b], Planware [Smi97], and Specware [WSGJ96].

The DTRE system performs data type refinements. A specification using abstract data types (sets, sequences, finite maps and tuples) is mapped to a specification with concrete data types (lists, bit-vectors, and array-based). The developer supplies some implementation directives, and DTRE completes the job by selecting the most efficient implementation based on those directives. The implemented system is then optimized based on the set of operations used as well as data flow analysis. The mappings from abstract to concrete types represent the design information, but unfortunately they are pre-built into the system and cannot be changed or added to by the developer. Additionally, the refinements are all of the type

Data-type → *Data-type*.

More complex data type restructuring involving multiple data types is not dealt with.

The KIDS system is designed for performing algorithm refinements. The developer creates a domain theory (an involved process) and then specifies a problem in terms of the domain theory. The problem must be written in the DRIO format (Domain type, Range type, Input predicate, Output predicate), where an element of the Domain type satisfying the Input predicate results in an element of the Range type satisfying the Output predicate. Based on the D and R portions of the problem to be solved the KIDS system offers a list of design tactics that may be suitable for solving the problem. A design tactic may have associated with it one or more algorithm and program schemes. After making choices the developer is prompted to fill in the parameterized algorithm and program schemes when necessary. The developer, using a number of transformation techniques (local optimizations, partial evaluation, finite differencing, etc.) can then optimize the resulting generated program code. The result is a function that, when given an element of type D satisfying precondition I, returns an element of type R satisfying post-condition O. The design information that KIDS uses is in the form of algorithm and program schemes, and the knowledge necessary to instantiate them. Part of this information is accessible to the developers, but much of it is hard-coded into the system, and cannot be extended or modified.

The Specware system is based on an algebraic specification language (sorts, operations, and predicate logic statements over the sorts and operations) that uses category theory concepts such as morphisms, interpretations, colimits, diagrams, etc., to construct and refine specifications [SJ95]. Individual specifications can be refined via an interpretation to other more concrete specifications. Such an interpretation is a piece of design information that may be generic enough to be applicable in a variety of situations. Constructing a diagram (a structure) of specifications and morphisms and then combining the diagram (via a colimit) can create a composite specification. If each individual specification in the structure has a refinement that is compatible with the other refinements then a refinement of the composite specification can be constructed automatically. The Specware system can represent design information for data type and algorithm design decisions but does not represent lower level design decisions as well.

The Planware system is a higher-level, domain-specific version of Specware designed to synthesize scheduling programs [Smi97]. Input to the system is accomplished by selecting characteristics of the tasks (release time, due time, quantity, and precedence, etc) and resources (consumable, and various forms of reusable) of some scheduling problem. The system then automatically generates a scheduling program via the same methodology used in KIDS except generalized to work in the Specware environment. The range of available data structures and implementation algorithms were carefully chosen by the synthesis system (actually the developers of the synthesis system) to be extremely efficient for the scheduling domain. While the general form of the data structures and algorithms are programmed into the synthesis system, the actual data structures and algorithms are created based upon the input specifications and problem constraints. The design information in Planware can be applied to create an entire family of scheduling applications but cannot be used to generate other types of programs.

Each of the systems described above is a research system that was built to solve realistic problems. Only the Specware system can be extend with additional design information. The control information that manipulates the design information and ensures that the parameterized information is correctly instantiated is non-existent in the Specware system and is hard-coded in the others.

2.1.6 Summary of refinement paradigms

The posit-prove paradigm has the greatest generality and has been of the most practical use in industry, but it has no explicit way to represent the higher-level design information. Consequently, automated support for the refinement process comes in the form of bookkeeping support and proof tools that help to prove the correctness of manually represented and applied design decisions.

The transformation approach has four variations: catalog rule-based, encoded rule-based, generative rule-based and refinement calculus based. The catalog rule and encoded rule transformation approaches have potentially no limit to the design information they can represent and apply. The problem is, the more complex the design information, the more complex the rule and the fewer situations to which it can be applied. The catalog rules are limited in that they are inherently syntax based. Catalog rules are useful for finishing touches, like peephole optimization in a compilation system and are often automatically applied. Encoded rules are typically less formal as they are procedural (read operational) in nature and may not be proved correct over the domains in which they can operate, otherwise they tend to have the same benefits and problems as the catalog rules. In both types of rule-based systems it is expected that the rules fire without much jittering and so they are limited to either low level design decisions or to specialized niche transformations.

The generative rule-based and refinement calculus-based approaches are limited in the type of design information that can be expressed. In their purest form, they are manually applied and all design information is emergent from the sequences of the application of the small transformations. There are some ways to package sequences of steps or to heuristically guide the use of the individual transformation steps that enables limited higher-level design information to be represented and applied. In addition, informal design tactics and approaches can be developed that when followed can help guide the developer in applying the rule set.

The deductive synthesis approach uses proof tactics instead of design information. The design information is one step removed from the refinement steps (which in this case are actually proof steps). In this approach, like the generative and refinement calculus approaches, the design information is implicit in a sequence of steps.

The classification synthesis approach can represent a variety of design information at a much higher level than the other refinement approaches. Examples include data type theories (sets as bags, sequences, arrays, bit vectors, etc. and various other mappings), algorithm theories (divide and conquer, global search, etc.), and application domain theories (scheduling, planning). The real work is done in advance when the higher-level design information is abstracted, generalized and proven correct. Applying the design information requires more work for a greater gain, as a larger granularity of design decisions can be made with the classification approach. Unfortunately, establishing a relationship between a requirement specification and design information that should apply to it may be arbitrarily difficult. The key to making classification synthesis more viable is increasing the ease in making the classification to begin with. In other words, increase the ability for design information and (parts of) a requirement specification to be related, and increase the breadth of design information that can be represented and applied.

Of those systems that could represent and apply design information in a formal manner, one family of systems, DTRE, KIDS, Planware and Specware [SJ95, SLM98], stood out as being able to represent and apply high level data structure design decisions and algorithm design decisions [Gra96,Smi90a,Smi97]. The theory underlying the representation and application of design information in this family of systems is directly exposed in the Specware system.

The Specware language can be viewed as being analogous to an assembly level programming language; necessary low level features are present in the language, but few higher-level concepts are present. Although many other specification systems use the concepts of specifications and specification morphisms, the Specware system in particular pioneered the concept that the relationships between specifications are as important as the specifications themselves and used them to provide an open and extensible framework for representing and applying design information. In theory, all of the design information captured by the DTRE, KIDS, and Planware systems can be modeled in Specware. In practice, the best ways to represent and manipulate design information in an algebraic framework are still being developed [Gra96]. Specifically the Specware system has a limited notion of a diagram of specifications and morphisms in terms of being able to construct, manipulate and refine them. Construction is based on listing the nodes and arcs and refinement is based on the refinement of individual specifications instead of a larger structure.

2.2 Terminology

This section briefly and informally covers some of the terms used in this dissertation. As a service to the reader, each term is formally defined in the associated appendix definition.

2.2.1 Category theory terms

A *category* (Definition A.1) is a mathematical structure consisting of a collection of objects and arrows between the objects where each object has an identity arrow, all successive arrows compose, and the composition is associative. A *diagram* (Definition A.6) is a select collection of objects and arrows of a particular category. To be a *commutative* diagram (Definition A.7), all arrows between the same source and target objects in the diagram must be equivalent. Arrows within a category can be *isomorphic* (Definition A.8), *monomorphic* (Definition A.9) or *epimorphic* (Definition A.10) depending on their relationship with other arrows in the category.

A *functor* (Definition A.11) maps the objects and arrows of one category to those of another such that identity and arrow compositions are preserved. A category can have *subcategories* (Definition A.13) and *full subcategories* (Definition A.14) which have only a subset of the parent categories objects and arrows.

A *cocone* (Definition A.16) is a collection of arrows from a diagram to another object where the arrows in the cocone commute when composed with the arrows within the diagram. A *colimit* (Definition A.17) is a minimal target object of a cocone, as every other cocone object of that diagram has an arrow to it from the minimal cocone object. A category that has a cocone object for all diagrams is termed *cocomplete* (Definition A.18). A *pushout* (Definition A.15) is a specialization of a colimit over a particular diagram form.

2.2.2 Specification and Model terms

A *specification* (Definition B.14) contains a set of sorts (abstract data types), a set of operations (abstract function headers), and a set of axioms over the sorts and operations. A *specification morphism* (Definition B.18) maps the signature elements of a source specification to those of a target specification such that the (translated) axioms of the source specification are theorems of the target specification. Specifications and specification morphisms are the objects and arrows of category *Spec* (Proposition B.19).

A *model* of a specification (Definition B.34) has a set of elements (carrier set) for each sort and a function for each operation where all axioms of the specification are satisfied by the model. All models of the target of a specification morphism can be reduced by the extra carrier sets and functions such that they are models of the source specification (Proposition B.39).

2.2.3 Specification development and refinement terms

A *conservative extension morphism* (Definition C.8) is one in which any model of the source specification can be extended with additional carrier sets and functions so that it is a model of the target specification. A *definitional extension morphism* (Definition C.12) is one in which that model extension is uniquely defined. The property of being a conservative or definitional extension morphism reflects across a pushout square (Proposition C.19).

An *interpretation* (Definition C.20) is a morphism from a source specification to a definitional extension (mediator) of a target morphism that indicates how a model of the target specification can be extended to become a model of the source specification. An *interpretation morphism* (Definition C.24) uses specification morphisms to link the source, mediator and target specifications of two interpretations such that the larger *Spec* diagram commutes. Interpretations and interpretation morphisms are the objects and arrows of category *Interp* (Proposition C.25).

Diagram refinement (Definition C.28) enables a source diagram to be refined by aggregating compatible interpretations from each of the objects in the source diagram.

2.3 Current Parameterization Research

The use of parameterization and instantiation is effective for creating large objects from smaller reusable objects. This section describes the current research on using parameterization with algebraic specifications to develop aggregate specifications.

Informally, a parameterized object is an object that has both fixed and variable parts. Instantiating a parameterized object (often called parameter passing) fixes the variable parts based on an actual parameter that meets the requirements of the variable part and enables the object to be used for a particular task. A weaker form of a parameterized object is the extensible object. An extension can be thought of as an uncontrolled instantiation. There is more freedom in extension and therefore there is more potential for misuse. It is the ability to be parameterized and instantiated (or extended) that enables reusable objects to be used in different contexts. Parameterization has many benefits:

- Parameterization enables the common parts of a collection of related objects to be constructed as a parameterized object once and then reused via instantiation multiple times.
- Once proven correct, the fixed part of the parameterized object will always be correct in each instantiation (assuming the instantiation follows certain rules). The effort spent ensuring that the fixed part of the parameterized object is correct can be amortized over the many uses (instantiations) of the parameterized object.
- It may be possible to limit parameter passing to "correct" actual parameters. If the possible actual parameter can be pre-limited to a set that meets some formal criteria, then the parameterized object is limited to correct instantiations only.
- A parameterized object is on a higher level of abstraction than the primitive objects from which it is built.
- Parameterization enables larger objects to be built in an easier fashion by incorporating previously developed parameterized objects in their construction.
- A collection of parameterized objects and the way in which the parameterized objects are joined to construct a larger object provide an explicit structure for the larger object.

Historically, parameterization in an algebraic specification language has enabled a specification to have one or more changeable parts that can be fixed in a variety of different ways, thus enabling it to be (re)used in a variety of different contexts. Previous research in parameterized algebraic specifications has focused on the parameterization of a single "body" specification by one [BG77, EL78, Gan83, EM85] or more [Hax89, Dim98] "formal parameter" specifications.

The two main classifications of algebraic specification parameterization in the literature are pushout-based parameterization and λ-style parameterization [BKL91, Wir90, and Gau93]. Pushout-based parameterization is referred to here as category-theory-based parameterization as this name is more inclusive for describing the actual research.

In category-theory-based parameterization the underlying theory of parameterization is based on specifications, morphisms and diagrams of specifications and morphisms. Instantiating a parameterized specification involves some categorical type operation such as taking a pushout, multi-pushout [JOE94], or colimit of the resulting diagram. There are many variations on the variable part of the parameterization including single parameter, multiple parameter, and parameterization by a diagram. Instantiations of the parameterized specification can be nested or recursive.

Unfortunately, current notions of category-theory-based parameterization have two basic flaws when used for developing and refining large structured specifications. The first problem is that all current notions of category-theory-based parameterization consider the body of the parameterization to be a single parameterized specification. This eliminates all structure in the object being parameterized. The second problem is that instantiation of a parameterized specification results in a specification and not a diagram. Thus the results of instantiation lack structure. The notion of parameterization and instantiation in current category theory based research lacks structure in two important dimensions and therefore is unsuitable for developing large structured specifications in a manner that makes the structure accessible for refinement.

In λ-style parameterization, a parameterized specification is a specification building function that is written in the form of a λ-expression [Wir90]. Instantiating a λ-style parameterization involves performing the specification operations on the actual parameters as required by the specification body. The actual parameter specification undergoes some kind of computable transformation or function application. Although the body of a λ-style parameterization is not a single specification (it is a collection of operations to be performed on the actual parameters), it is still limited in current research to producing a single specification as the result of instantiation. λ-style parameterization does not fit with the type of categorical operations developed in this dissertation and so will not be discussed further.

Appendix E describes current algebraic specification parameterization research in greater detail and provides examples of the various types of parameterization and instantiation.

2.4 Current Research on Constructing Diagrams

The Specware language [SJ95, SLM98] has a completely freeform means of combining specifications by listing the arcs and nodes of a diagram instead of by using some form of parameterization. As an example, the Specware language code in Figure 2-1 creates a *Spec* diagram whose colimit is a specification for a Map whose domain is Nat and whose codomain is Nat. Each specification in the node list becomes its own node in the diagram and if the same specification is needed for two different nodes it can be given two different "node names". Each morphism in the arc list "connects" two of the already given nodes. Although not depicted, the arcs can be named as well.

The Specware language diagram statement can be used to specify any finite diagram of specifications and morphisms. As such, single-parameterization, multi-parameterization, and parameterization by a diagram (Appendix E) can all be emulated using the Specware language. Nested and recursive instantiation can also be emulated. The term emulated is used because, unfortunately, each Specware diagram must be created from scratch from individual specifications and morphisms. The only operation available over diagrams in the Specware language is the colimit operation. Diagrams in the Specware language cannot truly be called parameterized objects as they cannot be constructed with fixed and variable parts. The individual specifications making up a diagram can be extended or reused in a different diagram. The colimit object of a diagram can be extended or used in a yet another diagram, but a diagram itself cannot be directly extended or parameterized.

```
Diagram Map-from-Nat-to-Nat is
  nodes Map,
        s1:One-Sort, s2:One-Sort,
        Dom:Empty, Cod:Empty
  arcs s1->Map: {X -> Dom},
       s1->Dom: {X -> Nat},
       s2->Map: {X -> Cod},
       s2->Cod: {X -> Nat}
end-diagram
```

```
Spec Map-from-Nat-to-Nat is
  Colimit of Map-from-Nat-to-Nat
```

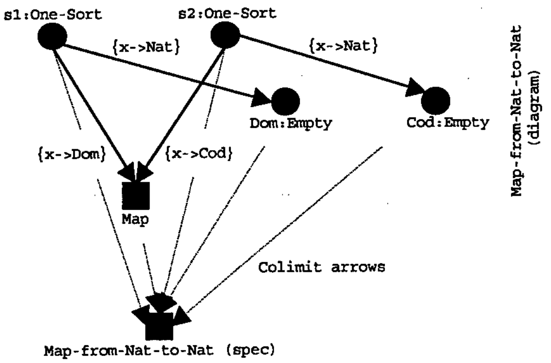


Figure 2-1. Specware language code and pictorial representation of a diagram and its colimit

If a parameterized object is viewed as the fixed and variable parts of a diagram, the specifier must (re)form both the fixed part and its variable instantiation within the same Specware diagram each and every time the "parameterized object" is to be instantiated. Because of this limitation, for example, creating an instantiated Map specification always requires the unintuitive diagram construction code depicted in the Specware language code for the diagram Map-from-Nat-to-Nat in Figure 2-1. The specifier must know and repeat the pattern of nodes and arcs in the diagram in order to extend (instantiate) the Map specification each time a Map of some sort to some other sort is needed. There is no means by which part of the work of creating the diagram can be accomplished and then stored in some fashion so it can be reused later.

Appendix D provides a larger example of developing a diagram using this diagram construction approach.

2.5 *Current Research on Refining Diagrams of Specifications*

Assuming that a structured requirement specification can be created, the problem then switches to developing a refinement for the aggregate specification. *Spec* interpretations (Definition C.20) are one of the primary definitions of refinement for specifications used by researchers [Ehr82, ST88, and Wir90]. Diagram refinement (Definition C.28) [SJ95, SLM98] enables interpretation refinements to be composed along with the specification components being refined to form an interpretation of the composite specification. Unfortunately, there are several limitations and misapplications of diagram refinement and interpretations as defined and applied in [SLM98].

This section describes each of the following limitations/misapplications of *Spec* interpretations and diagram refinement in greater detail.

- Diagram refinement using *Spec* interpretations as the design information to be applied cannot adequately represent higher-level design information, specifically "structured" design information, that involves one or more component specifications.
- Diagram refinement is limited to a refined target structure that is inextricably linked to the structure of the requirement specification; it does not allow refinements whose target diagram's structure is appreciably different than the source diagram's structure.
- Diagram refinement is used to refine "parameterized specifications," a use for which it is not suited.

2.5.1 *Spec* Interpretations vs. higher-level design information

Spec interpretations only represent specification to specification refinements; as such they cannot adequately represent more high-level structured design information that may involve the refinement of several "source" specifications at the same time.

Figure 2-2 depicts a collection of *Spec* interpretations (as indicated in the upper left corner) that purport to refine a Set-of-Nats into a Tree-of-Nats. Each of the three *Spec* interpretations, Nat ⇒ Nat, Set ⇒ Tree, and One-Sort ⇒ One-Sort, is individually a valid *Spec* interpretation. They can be individually applied to (matched with) the specifications in the source diagram and interpretation morphisms can be formed between the *Spec* interpretations (the triple arrows in Figure 2-2) such that a larger commuting diagram in *Spec* is formed. A colimit in *Interp* of the interpretations and interpretation morphisms results in an interpretation Set-of-Nat ⇒ Tree-of-Nat. From outward appearances, the diagram refinement is a success, yet the resulting *Spec* interpretation is not what is wanted.

The diagram refinement shown in Figure 2-2 is incorrect (or incomplete) because in order for a Tree to store elements in sorted order there must be a defined *and indicated* total order operation over the element sort of the Tree specification, in this case the sort Nat. The sort Nat has several such operations; among them are the "Less than" and "greater than" operations. However, the diagram refinement did not indicate which total order operation over the sort Nat should be used, and therefore the diagram that produced the Tree-of-Nat specification is incorrect. Storing design information as *Spec* interpretations does not allow such information to be specified as it is an emergent property of the collection of *Spec* interpretations making up the diagram refinement as opposed to any individual *Spec* interpretation.

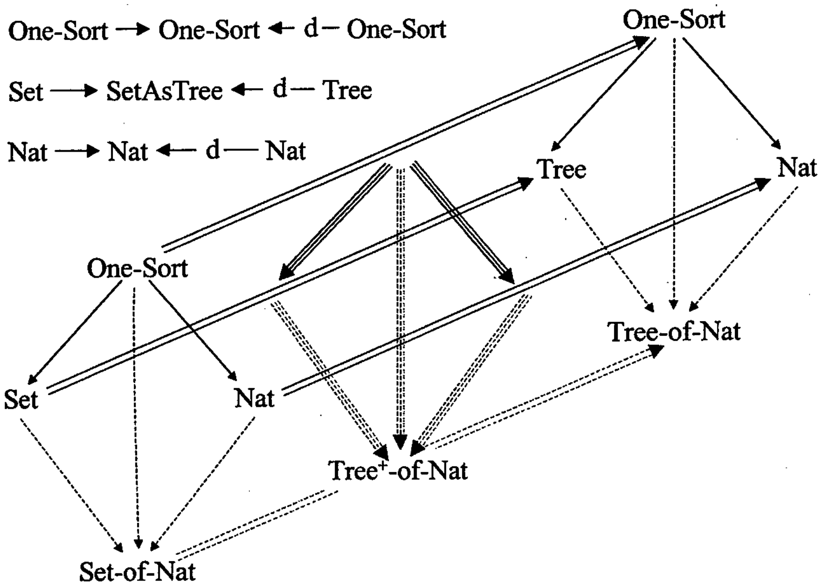


Figure 2-2. Incorrect collection of interpretations for a diagram refinement

Figure 2-3 depicts a correct diagram refinement of Set-of-Nat to Tree-of-Nat, as the One-Sort specification is refined via the *Spec* interpretation One-Sort ⇒ Total-Order instead of the *Spec* identity interpretation One-Sort ⇒ One-Sort. In Figure 2-3, when interpretation morphisms are developed between the *Spec* interpretations, the specification Total-Order ensures that the element sort and the total order operation over the element sort in the Tree specification will be combined with the Nat sort and the total order operation over Nats. This ensures that the target diagram of the diagram refinement, the specification Tree-of-Nat, and the resulting composite *Spec* interpretation, Set-of-Nat⇒Tree-of-Nat, is correct.

While diagram refinement as a whole can adequately handle the refinement depicted in Figure 2-3, representing design information as individual *Spec* interpretations cannot adequately represent the higher-level requirements that must be present for the collection of *Spec* interpretations to be what is wanted. It is the developer's knowledge that must be used to distinguish between the incorrect diagram refinement in Figure 2-2 and the correct diagram refinement in Figure 2-3 and not anything that can be specified using *Spec* interpretations and diagram refinement alone. In other words, correct and compatible (with each other) individual interpretation refinement choices do not always result in a correct overall choice, as additional design information beyond what is contained in a *Spec* interpretation is needed.

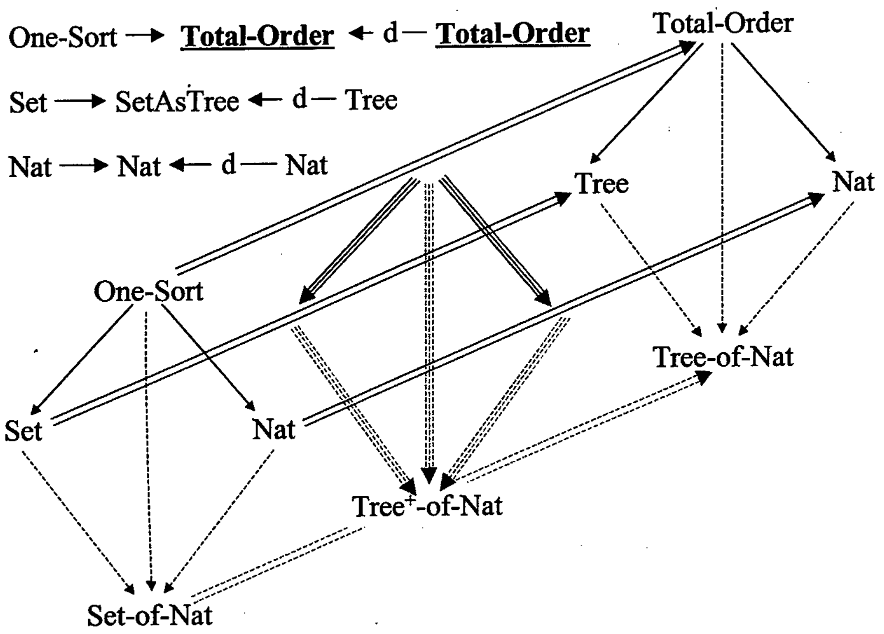


Figure 2-3. Correct collection of interpretations for a diagram refinement

Part of the problem is that specifications Set and One-Sort can be viewed linked together in a form of parameterization that may not be taken into account when selecting individual interpretations. In Figure 2-3 the formal parameter being refined is specification One-Sort, the body being refined is specification Set, and the actual parameter being refined is specification Nat. In [Sri97] an approach for refining parameterized specifications (referred to as pspecs in the reference) was developed that linked the refinements of the "formal parameter" specification and refinements of the "body" specification and allowed them to be refined as a unit. The pspec refinement technique described in [Sri97] did not describe how the refinement of pspecs fit within the larger context of a diagram in which it was embedded. Nor could it handle the case depicted in Figure 2-3 where the formal parameter of the target pspec (Total-Order) is required to be stronger (require more properties) than the formal parameter of the source pspec (One-Sort) as the refinement was independent of the actual parameter. In addition, the technique as described was limited to having a single specification as the body and a single specification as the formal parameter.

2.5.2 Restructuring refinements

Diagram refinement does not enable one to restructure the diagram while it is being refined, as a diagram refinement is made up of a collection of *Spec* interpretations that are inherently between a single source specification and a corresponding target specification. In diagram refinement, the structure of the requirement specification is directly reflected in the structure of the refinement and eventual implementation.

In [SST92] the authors differentiate between parameterized specifications and specifications of parameterized programs. Parameterized specifications are used to structure the requirement specifications. A related concept, the specification of parameterized programs, is used to specify the programming language modules that arise when designing an implementation, i.e. the structure of the implementation. The two structures, the requirement specification structure and the implementation structure, may be very different from each other depending on what design choices are made. If the structure of the requirement specification and implementation are identical, then there is no cause for concern, as diagram refinement uses the structure of the requirement specification to infer (or at least place a bound on) the structure of the implementation. However, if the desired implementation structure is (appreciably) different than the requirement specification structure, then interpretations and diagram refinement alone cannot be used to accomplish the refinement.

Figure 2-4 provides an example of the possible differences between a requirement structure and an implementation structure. If the requirement specification contains a set of pairs, as depicted on the left in Figure 2-4, then using diagram refinement and interpretations, one can choose from among a number of refinements for the individual specifications in the diagram. For example, the Set specification could be refined into Bags, Lists, Tree, Bit-vectors, etc., and the Pair specification could be refined into a record-type with two fields, a Nat field and a Flag field. However, there is no way to refine the diagram on the left in Figure 2-4 to the one on the right using interpretations and diagram refinement.

On the right is a Map from the first Pair element (A) to a Set of the second pair element (B). This Map-to-Set structure on the right can be definitionally extended with operations over the <A, B> pair so as to emulate a Set of Pairs. Thus the structure on the right can be extended so as to be an implementation of the structure on the left. Refining the diagram on the left to the diagram on the right in Figure 2-4 is not possible using diagram refinement.

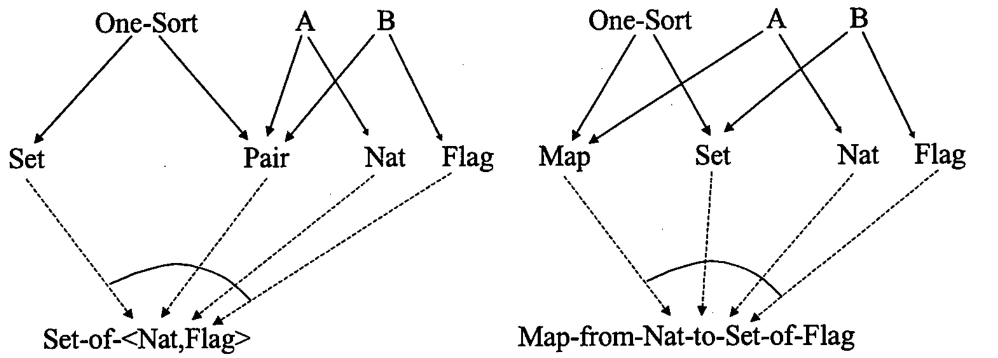


Figure 2-4. Restructuring refinement

However, given that one knows in advance that such a structured refinement is desired, one can develop the requirement specification as depicted in Figure 2-5 (or one can restructure the requirement specification to appear as depicted in Figure 2-5 whether or not it was developed that way initially). In Figure 2-5, the derived specification Set-of-Pair can be refined as a unit given a *Spec* interpretation Set-of-Pair ⇒ Map-to-Set. In other words, to apply the "structured" design information, one has to either develop the requirement specification with the particular design refinements in mind, or one must restructure the requirement specification once it is realized that a particular refinement is desired. In either case the effect is to remove the structure or "collapse" the structure prior to the refinement that is taking place (prior to the interpretation being applied).

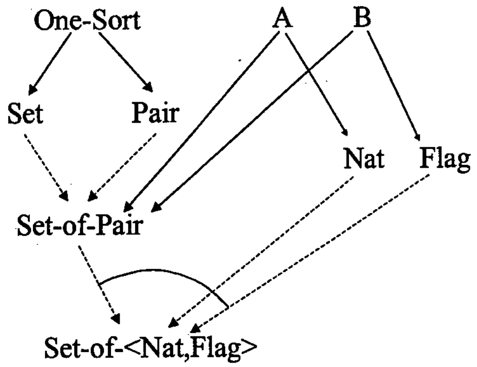


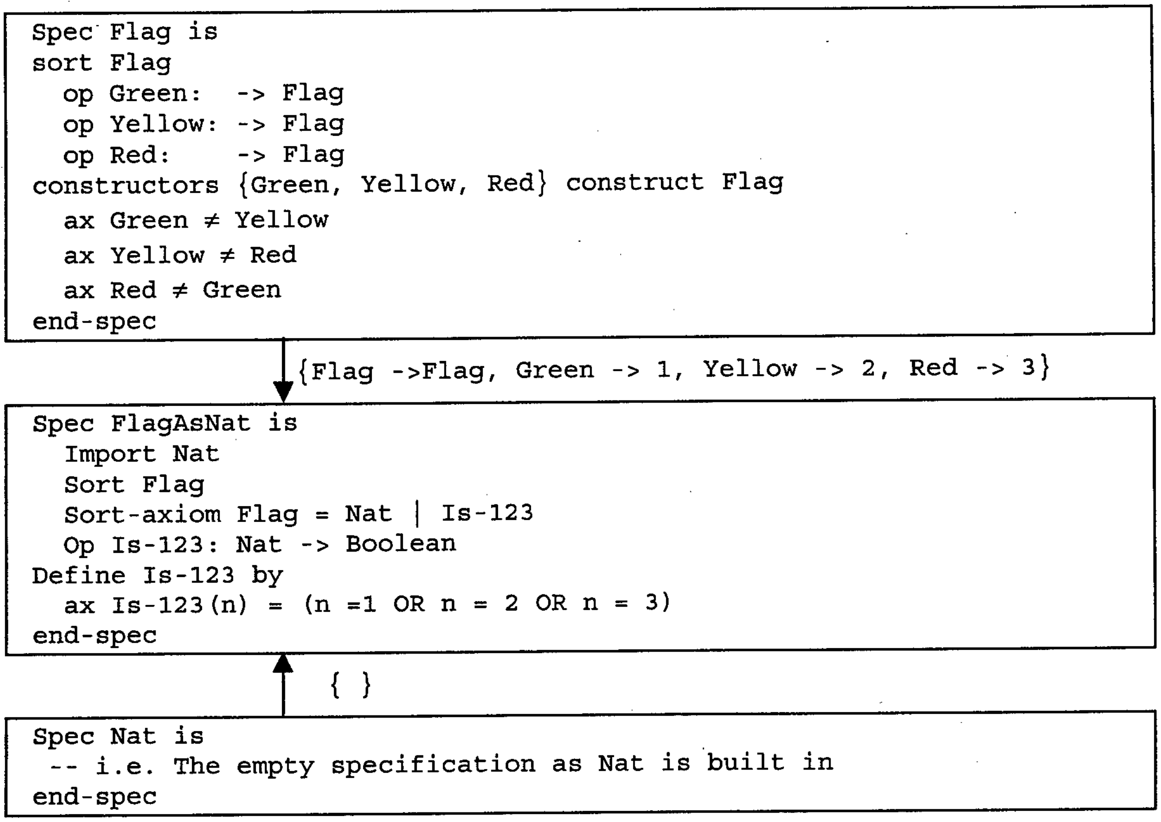
Figure 2-5. Multiple levels of requirement specification structure

Normally one applies an interpretation in diagram refinement by identifying the source specification of an interpretation with a specification in the diagram being refined. This can be accomplished mechanically via a simple search through a library of design refinements (i.e. *Spec* interpretations). How can one determine that the "collapsed" *Spec* interpretation source specification (i.e. the specification Set-of-Pair) can be applied to a requirement specification without first restructuring the requirement specification and why would one collapse the requirement specification without previously identifying that a particular *Spec* interpretation refinement could be applied to it? Again, *Spec* interpretations and diagram refinement alone are not up to the task.

2.5.3 Parameter vs. Body refinement

Certain specifications are called body, formal parameter, and actual parameter based on their relationship with each other; see Appendix E and Section 4.2.1. It ought to be possible to refine the actual parameter in a manner that is independent of the refinement of the body. In general, this is not possible in diagram refinement.

In the *Spec* interpretation in Figure 2-6, the Flag sort is implemented as a subsort of the Nat sort, namely the subsort containing the Nat values 1, 2, and 3. The constant Flag operations, Green, Yellow, and Red are implemented as the Nat values 1, 2, and 3. As the requirement specification is a Set of Flag, one ought to be able to refine the Flag specification via the interpretation Flag ⇒Nat depicted in Figure 2-6, independently of any refinement being done to the Set specification.



```
Spec Flag is
sort Flag
  op Green:  -> Flag
  op Yellow: -> Flag
  op Red:    -> Flag
constructors {Green, Yellow, Red} construct Flag
  ax Green ≠ Yellow
  ax Yellow ≠ Red
  ax Red ≠ Green
end-spec
```

{Flag ->Flag, Green -> 1, Yellow -> 2, Red -> 3}

```
Spec FlagAsNat is
  Import Nat
  Sort Flag
  Sort-axiom Flag = Nat | Is-123
  Op Is-123: Nat -> Boolean
Define Is-123 by
  ax Is-123(n) = (n =1 OR n = 2 OR n = 3)
end-spec
```

{ }

```
Spec Nat is
 -- i.e. The empty specification as Nat is built in
end-spec
```

Figure 2-6. Interpretation: Flag as Nat

Such a diagram refinement is depicted in Figure 2-7 where the *Spec* interpretations for the Set specification and the One-Sort specification are *Spec* identity interpretations and the *Spec* interpretation for the Flag specification is as depicted in Figure 2-6. Essentially, in the diagram refinement in Figure 2-7, the design choice is to refine the Flag sort in the Flag specification to a sub-sort of the Nat sort. The question, as indicated by the two "?"s in Figure 2-7, is how to form the required interpretation morphisms between the three interpretations making up the diagram refinement.

The diagram in Figure 2-8 is a flattened view of the diagram refinement pictured in Figure 2-7. It depicts the morphisms between the specifications making up the diagram refinement as well as (some of) the signature element mappings between the various specifications. Those signature element mappings indicate the needed interpretation morphisms that should form a commuting diagram. Unfortunately, a commuting diagram cannot be formed given the three interpretations listed in Figure 2-7.

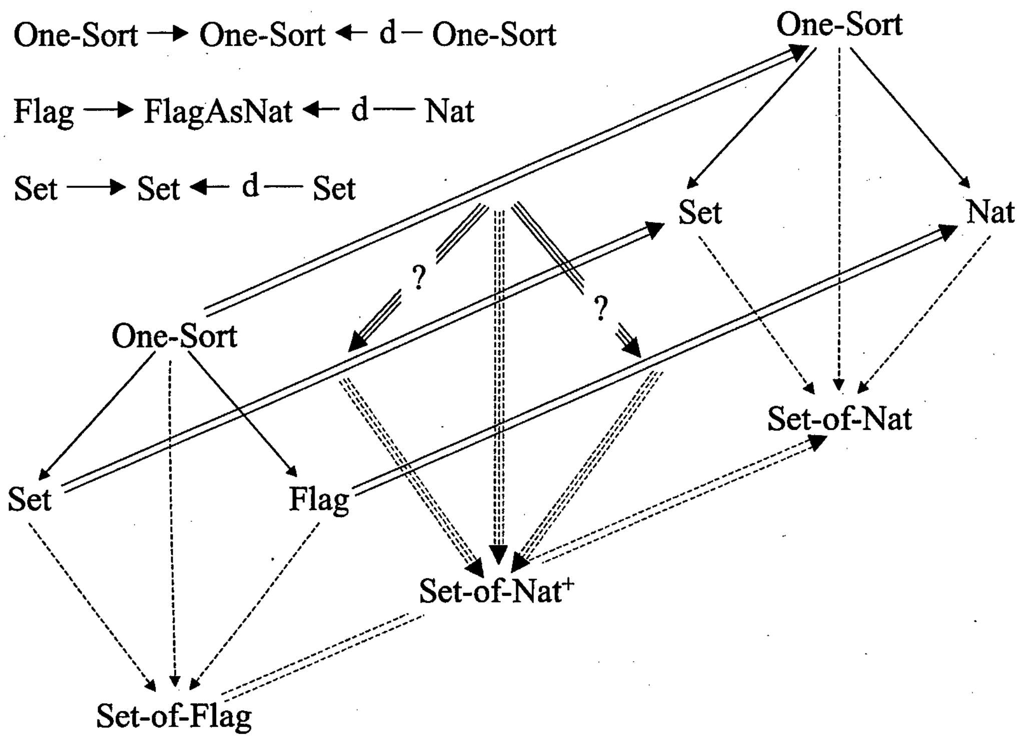


Figure 2-7. Diagram refinement of Set-of-Flag to Set-of-Nat

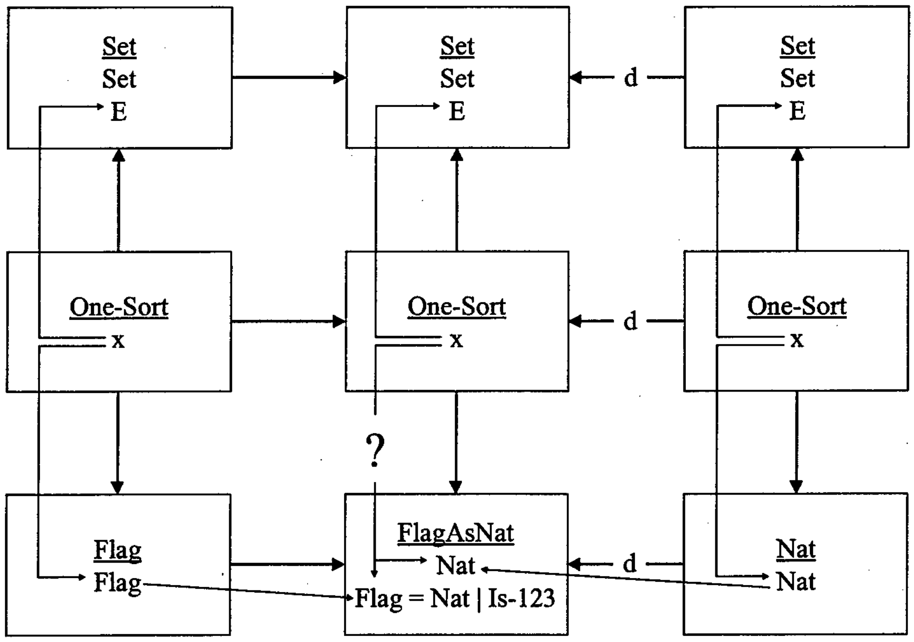
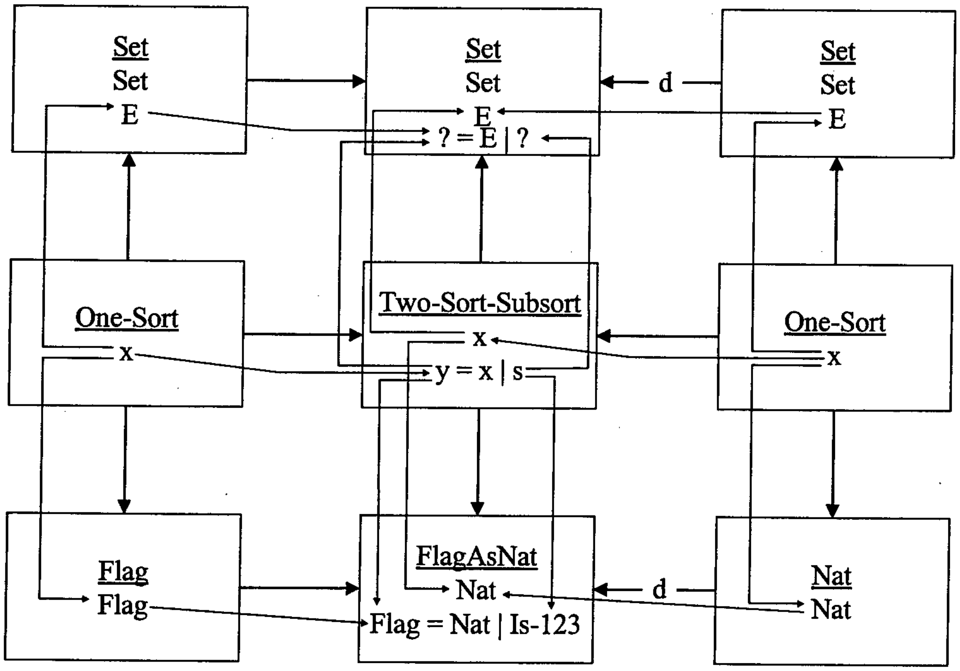


Figure 2-8. Required interpretation morphism cannot be formed

The "?" in Figure 2-8 indicates the problem. In order for the lower left square of specifications One-Sort, One-Sort, Flag, and FlagAsNat to be a commuting square, the sort X in the mediator One-Sort specification must be mapped to the sort Flag in the FlagAsNat specification. On the other hand, in order for the lower right square of specifications, One-Sort, One-Sort, FlagAsNat, and Nat to form a commuting square, the sort X in the mediator One-Sort specification must be mapped to the sort Nat in the FlagAsNat specification. Since both mappings cannot be possible, one or the other of the lower squares will not be a commuting square and the diagram in Figure 2-8 as a whole will not be a commuting diagram.

The solution to this dilemma (according to an example diagram refinement in [SLM98]) is to use interpretation schemes for the Set and One-Sort specifications rather than the *Spec* interpretations that are currently being used. An interpretation scheme is an "interpretation" where the target to mediator morphism is not a definitional extension [SLM98]. In Figure 2-9, the interpretation scheme One-Sort → Two-Sort-Subsort ← One-Sort is used to introduce an undefined subsort relationship in the mediator. (Hence the reason it is an interpretation scheme instead of an interpretation.) If one examines the bottom left and right squares in Figure 2-9, it is evident that these are commuting squares. Thus the problem depicted in Figure 2-8 has been solved. Unfortunately, the solution has introduced a new problem.



```
spec Two-Sort-Subsort
  sort x,y
  sort-axiom y = x | s
  op s: x → boolean
end-spec
```

Figure 2-9. Interpretation scheme used to partially fix the problem depicted in Figure 2-8

The newly introduced Two-Sort-Subsort mediator specification that solved the previous problem must also have a morphism to the Set mediator specification. This means that the sorts X and Y (and the binary relation S) in the Two-Sort-Subsort mediator specification must all be mapped to signature elements of the Set mediator specification. The sort X in specification Two-Sort-Subsort must be mapped to the Sort E in the Set specification in order for the upper right square of specifications to commute. In order for the upper left square of specifications to commute the sort Y must be mapped to some subsort of E in specification Set (and the binary relation S must be mapped to some binary relation over E). No such subsort or binary relation exists in the Set specification, therefore the "fix" depicted in Figure 2-9 is only a partial solution.

In order to correct the problem using diagram refinement, the Set⇒Set *Spec* identity interpretation must also be replaced with an interpretation scheme. This "final fix" to the problem is not depicted, but it involves developing a specification Two-Set-Subsort that has two copies of the Set specification (referred to here as "A" and "B" copies) and that will serve as the mediator of an interpretation scheme Set→Two-Set-Subsort←Set. In the Two-Set-Subsort specification, the element sort associated with the Set sort "B" must be defined to be a subsort of the element sort associated with the Set sort "A", and the operations over the Set sort "B" must all be defined in terms of the operations over the Set sort "A". The binary relation that defines the element subsort is itself undefined. (Which is why it is an interpretation scheme instead of in interpretation.) A similar problem and its fully coded solution can be found in [SLM98].

As can be seen in the progression from Figure 2-6 to Figure 2-9, a simple *Spec* interpretation of an "actual parameter" (Flag) that involves a sub-sort is propagated up to the "formal parameter" (One-Sort) and then ultimately propagated over to the "body" (Set). Not only is the refinement of the parameter specification *not* independent of the refinement of the body specification, but one cannot even refine the body specification with the identity interpretation while the actual parameter is being refined.

If the actual requirement specification were a Set-of-Set-of-Flags then the ripple effect would extend even further over to the "formal parameter" and "body" interpretations of the outer Set specification as well. If one had a more complex requirement specification such as the Petri Net specification in Appendix D, then theoretically a simple refinement such as Place ⇒ Nat could ripple to all the other specifications in the diagram. At least in the Set-of-Set-of-Flags case once the complex Set→Two-Set-Subsort←Set interpretation is formed it can be reused for both the inner and outer Set specification refinements. (Actually *non-refinements* since the Set specification is being refined to itself so one is not refining the Set specification at all.)

An alternate design choice for refining a Set-of-Flags would be to refine the Flag sort to be a quotient sort of Nat (i.e. all Nats equivalenced by their remainder divided by 3) instead of refining it to be a subsort of sort Nat. If this alternate design choice were chosen then the same ripple effect would apply, except now instead of rippling a subsort through the One-Sort and Set identity interpretations one would have to ripple a quotient sort through the identity interpretations and develop quotient sort interpretation schemes instead of subsort interpretation schemes. As a final complication, if one were to try to refine one of the Set specifications to a List specification at the same time (during the same diagram refinement step) as either of the Flag to Nat refinements were being accomplished, then the propagation of problems would be even more severe, if not impossible, to surmount.

The reason this refinement-rippling problem in diagram refinement occurs at all is that diagram refinement appears to be solving the wrong problem. The underlying meaning of the example diagram refinement Set-of-Flag to Set-of-Nat is that given an implementation of Set-of-Nat one can construct (via a definitional extension) an implementation of Set-of-Flag. Starting out with a Set-of-Nat and trying to develop a Set-of-Flag is not the problem that should be solved. Imagine that one had a Set module and natural numbers and that one was trying to develop a Set-of-Flag implementation. Would the first action be to develop a Set of natural numbers by instantiating the Set module with natural numbers and then secondly work with that instantiated Set-of-Nats module to develop a Set of Flags? Or would one first use the natural numbers to define the Flag data type and then secondly instantiate the Set module with the newly defined Flag data type ignoring the fact that a Set-of-Nat instantiation could be done at all? The latter choice is obviously the correct choice. However, since the goal of diagram refinement for this particular problem is to develop a *Spec* interpretation from the Set-of-Flag specification to the Set-of-Nat specification, it is the former choice that is the problem that is being solved by diagram refinement.

2.6 Summary

This chapter has described the existing research for representing and applying software design information by classifying and analyzing the various formal development methodologies. The latter sections of this chapter described specific problems with creating and refining large structured requirement specifications that occurs with existing approaches. The original research that solves these problems is presented in the following chapters.

3 Theory of Diagram Categories

This chapter develops the theory of diagram categories that underlies the category of diagrams of specifications proposed in Chapter 4. The material in Section 3.5 of this chapter defines a collection of operations over diagrams that are used in turn to define the syntax for creating and manipulating diagrams developed in Chapter 5. The colimit defined in this chapter underlies the application and composition of design information presented in Chapter 6.

The primary contribution of this chapter is the development of the diagram category and operations over diagrams. The diagram category, *d*X, is defined generically over any category X, instead of a specific category such as *Spec*, as this simplifies its development by eliminating extraneous details of the underlying category. This method of development also clearly demonstrates the relationship between certain properties and operations in the underlying category X and the corresponding properties and operations in the category *d*X.

Informally the objects of category *d*X are all the diagrams of a category X and the arrows of category *d*X are collections of X-arrows with certain structure and content-preserving properties. A *d*X-arrow, f_D:D_1→D_2, consists of an X-arrow from each X-object in the source *d*X-object (the X diagram D_1) to an associated X-object in the target *d*X-object (the X diagram D_2) that forms an even larger diagram in category X. Figure 3-1 depicts an example arrow f_D :D_1→D_2 in a category *d*X as well as a possible underlying structure of the objects and arrows in a category X.

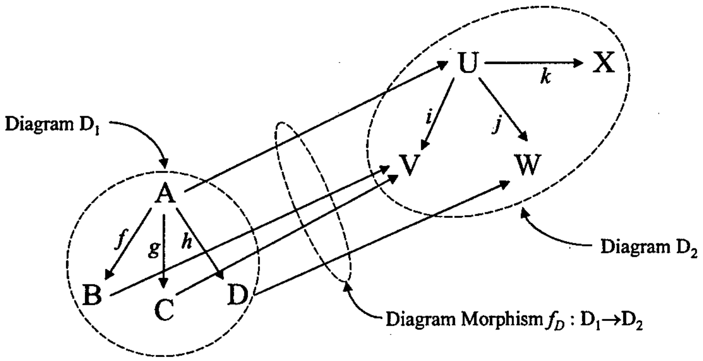


Figure 3-1. A morphism between diagrams

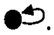
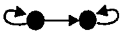
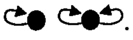
For (the collection of X-arrows) $D_1 \rightarrow D_2$ to be a diagram morphism the diagram structure of D_1 must be preserved in D_2 (via a structure-preserving mapping), the objects within D_1 must be preserved in D_2 (via an X-arrow mapping between the objects), and the arrows within D_1 must be preserved in D_2 (via the corresponding arrows forming a commutative square with the arrows between the diagrams). The category of shapes and the arrow category of category X are used to define these properties formally.

Sections 3.1 and 3.2 contain background information and "lemmas" necessary for later chapters. In Section 3.1 the category of shapes is formally defined and proved to be cocomplete. In Section 3.2 the arrow category, $X^{\rightarrow}$, of a category X is formally defined and some of the operations that are relevant to the category of diagrams being developed are defined. In Section 3.3 the notion of a morphism between diagrams is formally defined using the category of shapes and the Arrow category. Also in this section diagrams and diagram morphisms of any category X are proved to be the objects and arrows of a category *d*X. In Sections 3.4 and 3.5 the operations Flatten, Partition, Join, and Fold are defined as well as the notion of a diagram extension, diagram parameterization and diagram instantiation. In Section 3.6 a colimit operation is defined over Nice category *d*X diagrams (diagrams that flatten to commuting category X diagrams).

3.1 The Category of Shapes

An understanding of this section's material, especially the terminology, is critical for understanding Sections 3.3 through 3.6 as well as some of the later chapters. The proof that the category of shapes is cocomplete (Section 3.1.3) is original. The concept of a diagram being defined as a functor is common in category theory [AHS90]. The concept of using a category of shapes to capture and relate diagram structure information comes from [SJ95].

3.1.1 A shape category vs. the category of shapes

A shape category or a shape can be thought of as the structural essence of a category (the number and orientation of the objects and arrows in a category). For example, every category with a single object and a single arrow has the structure (shape): . This category is commonly referred to as Category **1** for its single object and single identity arrow [Gol84]. A category with two objects and three arrows must have one of the following two shapes:  or . As these shape abstractions have their own objects and arrows, and some of these arrows are identity arrows and these arrows compose and are associative (as defined by any of their underlying "concrete" categories), each shape is itself a category.

Definition 3.1. *Shape category*: The objects of a shape category are called nodes; the arrows of a shape category are called arcs, with each arc having a source and target node. Each node has an identified identity arc, and for every pair of arcs that form a "path", $\bullet_1 \to \bullet_2 \to \bullet_3$, there exists a composed arc $\bullet_1 \to \bullet_3$. The composition of arcs is associative. As noted in [AHS90], the definitions of a shape category (or scheme as it is referred to there) and a category are identical.

A shape category is useful for its structural properties. Note that the physical orientation of the shape category is unimportant. Thus the shape category as depicted in Figure 3-2 could be rotated, flipped or manipulated via some graph isomorphism and it would still be the same shape category.

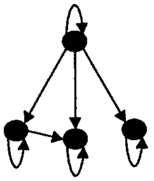


Figure 3-2. An example shape category

Functors between categories have a direct relationship with these shape categories. For example, each category with shape $\bullet\circlearrowleft$ has two functors to each category with shape $\circlearrowright\bullet\longrightarrow\bullet\circlearrowleft$, and two functors to each category with shape $\circlearrowright\bullet \quad \circlearrowright\bullet\circlearrowleft$. The reason a category of the latter shape has only two functors to it and not three is because each object can only have one identity arrow and functors must preserve identity arrows. Every category, regardless of its shape, has a functor to every category of shape $\bullet\circlearrowleft$. There are eight functors from each category with shape $\circlearrowright\bullet\longrightarrow\bullet\circlearrowleft$ to each category with the shape depicted in Figure A-3 (four that "preserve the separation of nodes" and four that do not). There are five functors from the shape depicted in Figure A-3 to shape $\circlearrowright\bullet\longrightarrow\bullet\circlearrowleft$. Each of these "always exists" functors is a morphism between shapes (and a functor between shape categories.) A functor between shape categories captures the structural essence of a functor between "concrete" categories and abstracts away the underlying meaning of the mapping of the objects and arrows between the "concrete" categories.

Definition 3.2. *Shape functor*: A shape functor is a functor between shape categories, i.e. any mapping of the nodes and arcs of one shape category to another such that identity arrows and arrow composition is preserved by the mapping.

Terminology: The terms shape and shape category are used interchangeably and the terms shape morphism and shape category functor are used interchangeably based on the context. The objects of a shape category are referred to as nodes and the arrows of a shape category are referred to as arcs.

Definition 3.3. *Category of shapes*: The objects of the category of shapes are all the shapes; the arrows of the category of shapes are all the shape morphisms. The category of shapes is a full sub-category (Definition A.14) of category *Cat*.

3.1.2 Classifying shape morphisms

Shape morphisms can be classified by the way they "preserve" the structure of a shape.

A shape *monomorphism* $\sigma:G_1 \rightarrow G_2$ associates each node and arc in shape G_1 with a distinct node and arc in shape G_2. A shape monomorphism preserves the shape "structure" of shape G_1 intact without any collapsing in shape G_2. However, a shape monomorphism does not rule out additional unmapped arcs and nodes from occurring in shape G_2.

In a shape *epimorphism* $\sigma:G_1 \rightarrow G_2$ every node and arc in shape G_2 is mapped to by at least one node and arc from shape G_1 thus the structure of G_2 is completely "covered" by the structure of G_1. However, more than one shape G_1 arc or node may be mapped to a single shape G_2 arc or node and thus the structure shape of G_1 may be collapsed in its mapping to shape G_2.

A shape *isomorphism* in the category of shapes neither extends nor collapses the structure of the source category in the target category. Essentially no change in structure has occurred based on a shape isomorphism.

3.1.3 Colimits in the category of shapes

In this section the category of shapes is proved to be cocomplete (Definition A.18) by first proving that it has all pushouts (Definition A.15).

Informally, a pushout in the category of shapes combines two shapes based on the sharing indicated by a third shape. A pushout diagram in the category of shapes, $G_{B1} \leftarrow\sigma_{B1}- G_A -\sigma_{B2}\rightarrow G_{B2}$, is a an object G_C, that merges shapes G_{B1} and G_{B2} based on the sharing indicated by shape G_A. A pushout object in the category of shapes preserves as much of the shape structure as possible of the shape objects making up the pushout diagram while not adding additional extraneous structure or combining more of the structure than is required.

Figure 3-3 depicts an example pushout in the category of shapes. It has been simplified in order to avoid cluttering the figure. Specifically, the identity arrows and arrow compositions within the individual shapes are deliberately not depicted. In addition, the arrows between the shape objects (the functors between the shape categories) depicts only the mapping of nodes to nodes; the mapping of arcs to arcs (in this case) can be uniquely determined from the node mappings. Finally, the arrow $G_A \rightarrow G_C$ has also been left off the figure, as it can easily be determined.

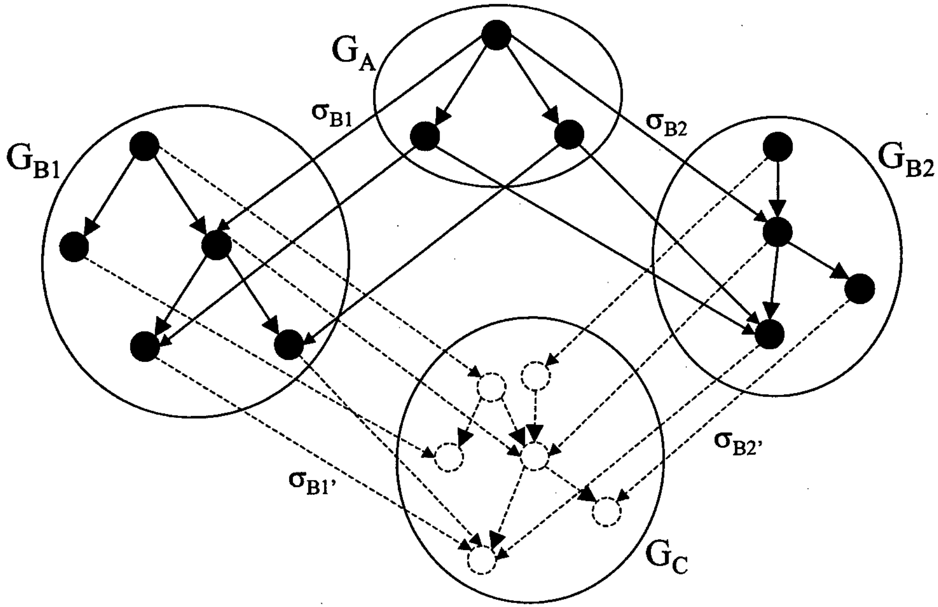


Figure 3-3. Underlying structure of a pushout in the category of shapes

Proposition 3.4. The category of shapes has all pushouts.

Proof:

Construction of a pushout object: Given a diagram in the category of shapes of the form $G_{B1} \leftarrow\sigma_{B1}- G_A -\sigma_{B2}\rightarrow G_{B2}$, the pushout object G_C is constructed as follows:

First, the nodes and arcs of shapes G_A, G_{B1} and G_{B2} are labeled so as to be distinct from each other. This has been done for the nodes only in the example pushout diagram in Figure 3-4. (Ignore the labels in the pushout object G_C for now.)

Second, the shape morphism σ_{B1} is used to derive a collection of equivalent groups of nodes and arcs over its domain and codomain objects G_A and G_{B1}. The equivalence groups are defined as follows:

For each node $\eta \in \text{Objects}(G_{B1})$, the equivalence group $\eta_{\mathbf{B1}}$ is defined as follows:

$$\eta_{\mathbf{B1}} = \eta \cup \{ \alpha \mid \alpha \in \text{Objects}(G_A) \wedge \sigma_{B1}(\alpha) = \eta\}$$

For each arc $f \in \text{Arrows}(G_{B1})$, the equivalence group $\boldsymbol{f}_{\mathbf{B1}}$ is defined as follows:

$$\boldsymbol{f}_{\mathbf{B1}} = f \cup \{ g \mid g \in \text{Arrows}(G_A) \wedge \sigma_{B1}(g) = f \}$$

Thus, each $\eta \in \text{Objects}(G_{B1})$ has an associated equivalence group $\eta_{\mathbf{B1}}$ that contains one G_{B1} object and zero or more G_A objects. Each $f \in$ Arrows (G_{B1}) has an associated equivalence group $\boldsymbol{f}_{\mathbf{B1}}$ that contains one G_{B1} arrow and zero or more G_A arrows.

Similarly the shape morphism σ_{B2} is used to develop a collection of equivalent groups of nodes ($\eta_{\mathbf{B2}}$) and of arcs ($\boldsymbol{f}_{\mathbf{B2}}$) associated with the nodes (η) and arcs (f) of shape object G_{B2}.

Third, the equivalence groups associated with the arcs and nodes of shapes G_{B1} and G_{B2} are then merged based on the common nodes and arcs of shape G_A to form a merged collection of equivalent nodes, $\{\eta_C\}$, and arcs, $\{\boldsymbol{f}_C\}$, defined as follows:

Let the $|\mathbf{x}|$ operation be the size operation over an equivalent group $\mathbf{x}$,
Let $\boldsymbol{\sigma}_{\mathbf{B1}}$ be the mapping from a node $\eta \in \text{Objects}(G_A)$ or arc $f \in \text{Arrows}(G_A)$ to the associated equivalence group of G_{B1} of which it is a member,
Let $\boldsymbol{\sigma}_{\mathbf{B2}}$ be the mapping from a node $\eta \in \text{Objects}(G_A)$ or arc $f \in \text{Arrows}(G_A)$ to the associated equivalence group of G_{B2} of which it is a member,

$$\{\eta_C\} = \{\eta_{\mathbf{B1}} \mid \eta \in \text{Objects}(G_{B1}) \text{ and } |\eta_{\mathbf{B1}}| = 1\} \cup \{\eta_{\mathbf{B2}} \mid \eta \in \text{Objects}(G_{B2}) \text{ and } |\eta_{\mathbf{B2}}| = 1\} \cup \{\eta_{\mathbf{merge}} \mid \eta_{\mathbf{merge}} = \sigma_{\mathbf{B1}}(\eta) \cup \sigma_{\mathbf{B2}}(\eta) \text{ where } \eta \in \text{Objects}(G_A) \},$$

$$\{\boldsymbol{f}_C \} = \{\boldsymbol{f}_{\mathbf{B1}} \mid f \in \text{Arrows}(G_{B1}) \text{ and } |\boldsymbol{f}_{\mathbf{B1}}| = 1\} \cup \{\boldsymbol{f}_{\mathbf{B2}} \mid f \in \text{Arrows}(G_{B2}) \text{ and } |\boldsymbol{f}_{\mathbf{B2}}| = 1\} \cup \{\boldsymbol{f}_{\mathbf{merge}} \mid \boldsymbol{f}_{\mathbf{merge}} = \sigma_{\mathbf{B1}}(f) \cup \sigma_{\mathbf{B2}}(f) \text{ where } f \in \text{Arrows}(G_A) \}$$

Thus, each node equivalence group in the set $\{\eta_C\}$ is either a singleton node equivalence group from the shapes G_{B1} and G_{B2} or is a merged equivalence group from those shapes based on the common nodes from shape G_A. Each arc equivalence group in the set $\{\boldsymbol{f}_C\}$ is similarly characterized over the arc equivalence groups of shapes G_A, G_{B1}, and G_{B2}.

Finally, each node equivalence group in $\{\eta_C\}$ becomes a single node in the pushout object G_C and each arc equivalence group in $\{\boldsymbol{f}_C\}$ becomes a single arc in the pushout object G_C. Each node $\eta \in \text{Objects}(G_C)$ has a unique associated equivalence group η_C in the constructed set $\{\eta_C\}$ and each arc $f \in \text{Arrows}(G_C)$ has a unique associated equivalence group $\boldsymbol{f}_C$ in the constructed set $\{\boldsymbol{f}_C\}$. For each $f \in \text{Arrows}(G_C)$, $\text{dom}(f)$ and $\text{cod}(f)$ are defined as follows:

$$\text{dom}(f) = \eta \mid \eta \in \text{Objects}(G_C) \wedge \eta_C = \text{dom}(\boldsymbol{f}_C)$$
$$\text{cod}(f) = \eta \mid \eta \in \text{Objects}(G_C) \wedge \eta_C = \text{cod}(\boldsymbol{f}_C)$$

where $\text{dom}(\boldsymbol{f}_C) = \{\alpha \mid \alpha = \text{dom}(g) \text{ for some } g \in \boldsymbol{f}_C \}$,
i.e. the set of domain objects of the set of arrows in the arc equivalence group $\boldsymbol{f}_C$.

where cod(f_C) = {α | α = cod(g) for some $g \in f_C$ },
i.e. the set of codomain objects of the set of arrows in the arc equivalence group f_C.

The predicates dom(f_C) ∈ { η_C } and cod(f_C) ∈ { η_C } are guaranteed by the construction of the sets {η_C} and {f_C}.

The constructed object is a shape category: The collection of nodes and arcs making up G_C are a shape category as each node has an identity arc, all adjacent arcs have an associated composite arc, and composition is associative.

For each $\eta \in$ Objects(G_C) there exists an $f \in$ Arrows(G_C) such that dom(f_C) = η_C = cod(f_C)),
i.e. $f = id_\eta$, the identity arc of node η.
For each $f, g \in$ Arrows(G_C), $g \circ f \in$ Arrows(G_C)
because the individual arrows comprising the arc equivalence groups f_C and g_C that have an adjacent node also compose and that collection of compositions will also be equal to an existing arc equivalence group.

The shape morphisms σ_{B1}':$G_{B1} \rightarrow G_C$ and σ_{B2}':$G_{B2} \rightarrow G_C$ are derived from the mapping that takes each arc and node in G_{B1} and G_{B2} (respectively) to the associated equivalence group in shape G_C of which it is a member. Shape object G_C is part of a commuting square by construction of the morphisms σ_{B1}' and σ_{B2}'. The constructed object G_C always exists as this method of construction will work with any shape category diagram $G_{B1} \xleftarrow{\sigma_{B1}} G_A \xrightarrow{\sigma_{B2}} G_{B2}$.

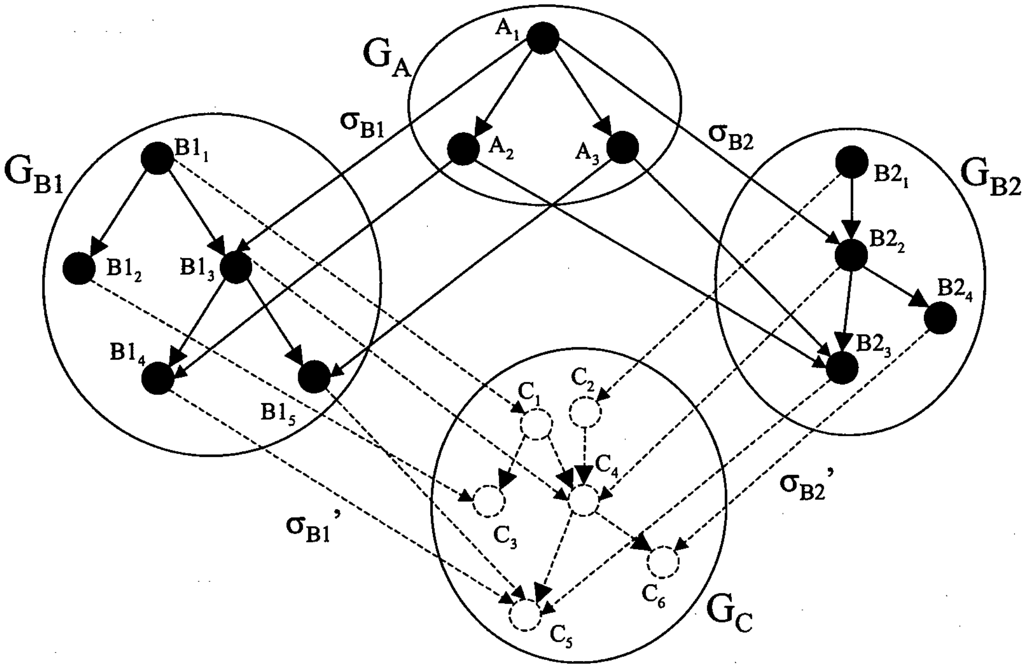


Figure 3-4. A distinct node labeling for a pushout in the category of shapes.

As an example of the pushout object construction, the shape morphism $\sigma_{B1}:G_A \rightarrow G_{B1}$ in Figure 3-4 induces the following node equivalent groups: $\{B1_1\}$, $\{B1_2\}$, $\{A_1, B1_3\}$, $\{A_2, B1_4\}$, $\{A_3, B1_5\}$. For shape morphism $\sigma_{B2}:G_A \rightarrow G_{B2}$, the induced collection of node equivalence groups is $\{B2_1\}$, $\{A_1, B2_2\}$, $\{A_2, A_3, B2_3\}$, $\{B2_4\}$. Combining the two collections of node equivalence groups by merging groups with common elements yields the following: $\{B1_1\}$, $\{B1_2\}$, $\{B2_1\}$, $\{A_1, B1_3, B2_2\}$, $\{A_2, A_3, B1_4, B1_5, B2_3\}$, $\{B2_4\}$. These equivalence groups are labeled C_1 through C_6 respectively in the pushout object G_C in Figure 3-4.

A similar process combines the induced arrow equivalence groups to form the arrows of shape G_C in Figure 3-4. The functors σ_{B1}' and σ_{B2}' map each object and arrow in shape categories G_{B1} and G_{B2} (respectively) into the equivalence group in G_C of which it is a member.

<u>Minimality of object G_C</u>: Assume that there exists another shape category, G_X, that forms a commuting square with the diagram $G_{B1} \xleftarrow{\sigma_{B1}} G_A \xrightarrow{\sigma_{B2}} G_{B2}$. Let each arc and node of shape G_X have an equivalence group consisting of itself and the arcs and nodes of the shape categories G_A, G_{B1}, and G_{B2} that are mapped to it by the cocone functors, $\mathcal{A}:G_A \rightarrow G_X$, $\mathcal{B}1:G_{B1} \rightarrow G_X$, and $\mathcal{B}2:G_{B2} \rightarrow G_X$.

$$\begin{aligned}
&\text{For each } \eta \in \text{Objects}(G_X),\ \eta_X = \{\ \eta\ \} \cup \{\alpha \mid \alpha \in \text{Objects}(G_A) \wedge \mathcal{A}(\alpha) = \eta\} \cup \\
&\qquad\qquad \alpha \in \text{Objects}(G_{B1}) \wedge \mathcal{B}1(\alpha) = \eta\} \cup \\
&\qquad\qquad \alpha \in \text{Objects}(G_{B2}) \wedge \mathcal{B}2(\alpha) = \eta\} \\
&\text{For each } f \in \text{Objects}(G_X), f_X = \{f\} \cup \{g \mid g \in \text{Arrows}(G_A) \wedge \mathcal{A}(g) = \eta\} \cup \\
&\qquad\qquad g \in \text{Arrows}(G_{B1}) \wedge \mathcal{B}1(g) = \eta\} \cup \\
&\qquad\qquad g \in \text{Arrows}(G_{B2}) \wedge \mathcal{B}2(g) = \eta\}
\end{aligned}$$

By construction of shape G_C, the equivalence groups associated with the arcs and nodes of G_C, and the equivalent groups associated with the nodes and arcs of shape G_X, the following two predicates will always hold:

For each $\eta_1 \in \text{Objects}(G_C)$ there exists an $\eta_2 \in \text{Objects}(G_X)$ such that $\eta_{1C} \subseteq \eta_{2X}$,
i.e. the equivalence group associated with η_1 is a subset of the equivalence group associated with η_2.
For each $f_1 \in \text{Arrows}(G_C)$ there exists an $f_2 \in \text{Arrows}(G_X)$ such that $f_{1C} \subseteq f_{2X}$
i.e. the equivalence group associated with f_1 is a subset of the equivalence group associated with f_2.

This subset relationship forms the basis of the universal arrow $G_C \rightarrow G_X$ that associates a G_X node with each G_C node and a G_X arc with each G_C arc. A G_C arc (node) can only be mapped to a single G_X arc (node) by this construction because the nodes (arcs) in the equivalence group associated with the G_C node (arc) must all be mapped to the same G_X node (arc) for G_X to be part of a commuting diagram.

If shape G_X contains additional arcs and nodes (additional "structure") not mapped to by the nodes and arcs of shape categories G_A, G_{B1}, and G_{B2}, then the equivalence group names of these additional arcs and nodes will contain their own names and no others and thus will not be mapped to by any nodes and arcs in shape G_C. Several G_C arcs and nodes could be mapped to the same G_X arcs and nodes if shape G_X has "collapsed" more of its shape than is necessary as indicated by object G_A and morphism σ_{B1} and σ_{B2}.

The mapping between G_C and G_X must form a shape morphism because the situation of an arc and its adjacent nodes in shape G_C being mapped to an arc and non-adjacent nodes in G_X would require such an occurrence to also happen between object G_A, G_{B1}, or G_{B2} and object G_X. This could not happen, as object G_X is assumed to be part of a commuting diagram.

<u>Universal arrow completes a commuting diagram</u>: The arrow $G_C \rightarrow G_X$ is part of a commuting diagram connecting the two commuting squares as the equivalence group names associated with the arcs and arrows of shape G_C must be a subset of those of shape G_X. This subset relationship indicates that the mapping of the arcs and arrows commute in the context of the larger diagram.

<u>Universal arrow is unique</u>: The arrow $G_C \rightarrow G_X$ is unique as a different arrow between the objects would necessarily violate the subset relationship among the equivalence group names and therefore would necessarily not be part of a commuting diagram □

Intuitively the construction of shape G_C is the maximal combined shape that does not introduce additional structure. If shape G_X has more structure than shape G_C then there is a unique arrow $G_C \rightarrow G_X$ as shape G_C can be "embedded" in shape G_X by not mapping G_C nodes and arcs to those extra nodes and arcs in G_X. If shape G_X combines ("collapses") shapes G_{B1} and G_{B2} to a greater degree than is needed based on the sharing required by shape G_A then shape G_X will have a unique arrow to it from shape G_C that collapses those same nodes and arcs.

Proposition 3.5. The category of shapes is cocomplete.

Proof: The empty shape (no nodes or arcs) is the initial object in the category of shapes as it has a unique arrow mapping its nodes and arcs (all *none* of them) to the nodes and arcs of any other shape. Any category that has an initial object and pushouts for all diagrams of the form B1←A→B2 is cocomplete [Gol84]. □

3.2 The Arrow Category, $X^{\rightarrow}$

The arrow category, $X^{\rightarrow}$, of category X is used in this thesis to express the property that the collection of X-arrows making up a morphism between diagrams preserves the objects and arrows of the source diagram in the target diagram. The material in this section supports the definition of a diagram morphism in Section 3.3 and the definition of a diagram category in Section 3.3.3. The *Source* and *Target* functors developed in Section 3.2.1 and the • operation in Section 3.2.2 are original but are based on similar functors and a similar operation presented in [SJ95].

An arrow category [Gol84, AHS90] is defined over an existing category using the arrows of that category as its objects and pairs of arrows that form commuting squares as its arrows.

Definition 3.6. Given a category X, the arrow category $X^{\rightarrow}$ has as its objects all of the arrows of category X, i.e. Objects($X^{\rightarrow}$) = Arrows(X). An arrow in category $X^{\rightarrow}$ is a pair of arrows of category X that form a commutative square with the domain and codomain objects of the $X^{\rightarrow}$-arrow, i.e. given $X^{\rightarrow}$-objects x:A→U and y:B→V an $X^{\rightarrow}$ arrow α:x→y is a pair of X-arrows f:A→B and i:U→V such that the resulting square commutes (See Figure 3-5). (Note that because the square commutes there is also a category $X^{\rightarrow}$ arrow <x, y>:f→ i.)

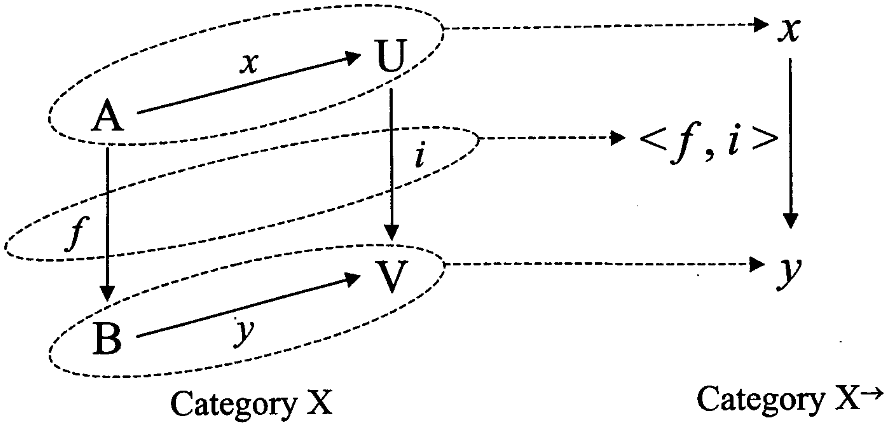


Figure 3-5. Category X compared with the arrow category $X^{\rightarrow}$

The objects and arrows of category $X^{\rightarrow}$ are completely defined by the arrows of category X, hence the name $X^{\rightarrow}$. An identity arrow in $X^{\rightarrow}$ of an object id_A is simply the pair < id_A, id_A>. Figure 3-6 depicts the composition of arrows in $X^{\rightarrow}$.

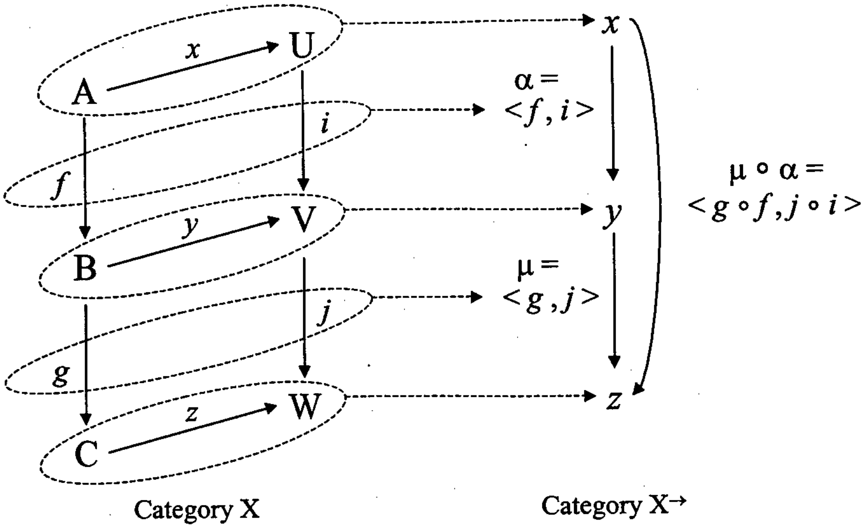


Figure 3-6. Sequential composition of X→ arrows

3.2.1 Functors *Source* and *Target* over the arrow category

The functors *Source*:X→→X and *Target*:X→→X are used in the definition of a morphism between diagrams in Section 3.3.

Definition 3.7. *Source* is a pair of functions < $Source_{Object}$, $Source_{Arrow}$ > that map category X→ objects and arrows respectively to category X objects and arrows. $Source_{Object}$ and $Source_{Arrow}$ are defined as follows:

for each $\alpha \in$ X→-objects where α:A→B, $Source_{object}(\alpha) = A$,
for each $f \in$ X→-arrows, where $f = <f_1, f_2>$, $Source_{Arrow}(f) = f_1$.

Definition 3.8. *Target* is a pair of functions < $Target_{Object}$, $Target_{Arrow}$ > that map category X→ objects and arrows respectively to category X objects and arrows. $Target_{Object}$ and $Target_{Arrow}$ are defined as follows:

for each $\alpha \in$ X→-objects where α:A→B, $Target_{object}(\alpha) = B$,
for each $f \in$ X→-arrows, where $f = <f_1, f_2>$, $Target_{Arrow}(f) = f_2$.

Proposition 3.9. *Source* and *Target* are functors.

Proof: An identity arrow in X→ for object α:A→U is the X→ arrow id_α: $\alpha \xrightarrow{<id_A,\, id_U>} \alpha$. Identity arrows are preserved by *Source* and *Target* as $Source_{Arrow}(id_\alpha)$ is the category X identity arrow for $Source_{Object}(\alpha)$ and $Target_{Arrow}(id_\alpha)$ is the Category X identity arrow for $Target_{Object}(\alpha)$. Functors *Source* and *Target* preserve the composition of arrows as the composition of X→ arrows $g \circ f$ where $f = <f_1, f_2>$ and $g = <g_1, g_2>$ is $g \circ f = <g_1 \circ f_1, g_2 \circ f_2>$ and therefore $Source(g) \circ Source(f) = Source(g \circ f)$ and $Target(g) \circ Target(f) = Target(g \circ f)$. □

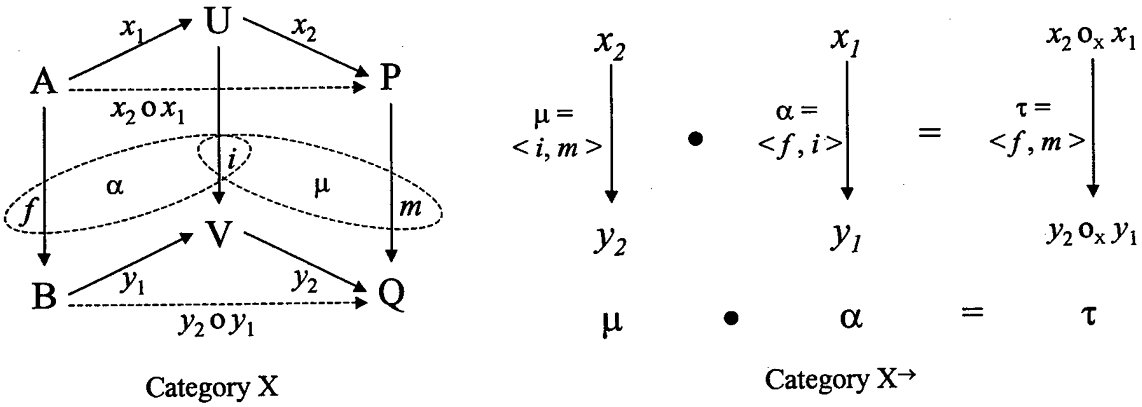
3.2.2 Parallel composition operation, •, over X$^{\rightarrow}$-arrows

Two "sequential" X$^{\rightarrow}$-arrows α:x→y and μ:y→z can be composed using the category X$^{\rightarrow}$ composition operation, $\circ$, as depicted in Figure 3-6. The • operation is a parallel form of composition, illustrated in Figure 3-7. The • operation can be thought of as the sequential composition operation "rotated 90 degrees"; i.e. instead of the commuting squares being one after the other (based on the X$^{\rightarrow}$-arrow direction), they are side by side. This second form of composition is used in Section 3.3.3 to prove that successive diagram morphisms compose.

Definition 3.10. The parallel composition operation, •, composes two X$^{\rightarrow}$-arrows α and μ, where *Target*(α) = *Source*(μ), to a third X$^{\rightarrow}$-arrow τ in the following manner:

$$\begin{aligned}\tau &= <\mathit{Source}(\alpha), \mathit{Target}(\mu)> \\ \mathrm{dom}(\tau) &= \mathrm{dom}(\mu) \circ_X \mathrm{dom}(\alpha), \text{ and} \\ \mathrm{cod}(\tau) &= \mathrm{cod}(\mu) \circ_X \mathrm{cod}(\alpha),\end{aligned}$$

where $\circ_X$ is the category X composition operation.



Category X

Category X$^{\rightarrow}$

Figure 3-7. Parallel composition of X$^{\rightarrow}$ arrows

Using the • operation to compose two X$^{\rightarrow}$-arrows, α and μ, where *Target*(α) = *Source*(μ) composes commuting squares in the underlying category X to form a new X$^{\rightarrow}$ arrow. The use of the sequential, $\circ$, and parallel, •, composition operations satisfy an interchange law [SJ95]: given four compatible category X$^{\rightarrow}$ arrows (α_1, α_2, β_1, $\beta_{2)}$ as depicted on the left in Figure 3-8, the equation on the right is true.

$$\begin{tikzcd}
x_1 \arrow[rr] & {} \arrow[d, "\alpha_1"'] & x_2 \arrow[rr] & {} \arrow[d, "\alpha_2"'] & x_3 \\
y_1 \arrow[rr] & {} \arrow[d, "\beta_1"'] & y_2 \arrow[rr] & {} \arrow[d, "\beta_2"'] & y_3 \\
z_1 \arrow[rr] & {} & z_2 \arrow[rr] & {} & z_3
\end{tikzcd}$$

$$(\beta_2 \bullet \beta_1) \circ (\alpha_2 \bullet \alpha_1) = (\beta_2 \circ \alpha_2) \bullet (\beta_1 \circ \alpha_1)$$

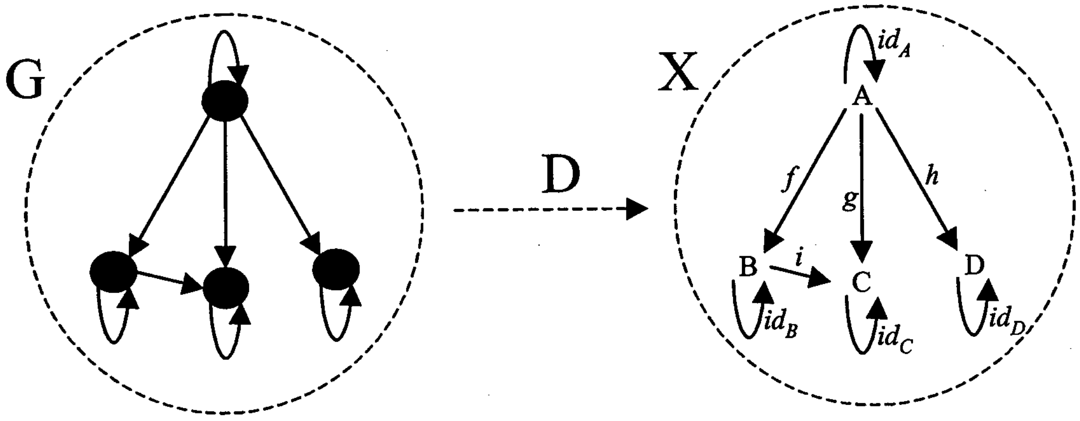
Figure 3-8. Interchange law of sequential and parallel composition operations

3.3 *The Category of Diagrams*

This section formally defines the notion of a diagram and a diagram morphism and proves that they are the objects and arrows of a category, *d*X. The definition of a diagram morphism and the definition of category *d*X are major contributions of this chapter and are highly important to this dissertation as they are the foundation for the rest of the document.

3.3.1 Diagrams

Recall that a diagram of category X is a functor D, from a shape category G, to category X, D:G→X [AHS90], See Figure 3-9.



a. Shape Category

b. Select objects and arrows from Category X that are mapped to by functor D

Figure 3-9. A diagram is a functor whose domain is a shape category

Definition 3.11. A diagram D of category X is a functor $< D_{Object}, D_{Arrow} >: G \rightarrow X$ from a shape category G to category X that maps the nodes(objects) and arcs(arrows) of a shape category G to selected objects and arrows of category X.

Notation: Given a diagram $D:G \rightarrow X$, and given a node (or arc) α in G, $D(\alpha)$ is the associated object (or arrow) in X. When the objects and arrows of D are referenced, this means the objects and arrows of category X that are mapped to by the functor D. Thus $\text{Objects}(D) = \{D(\alpha) \mid \alpha \in \text{Objects}(G)\}$ and $\text{Arrows}(D) = \{D(f) \mid f \in \text{Arrows}(G)\}$

Notation: An alternate method for defining a diagram exists where the objects and arrows of the diagram are "directly" listed and then the shape and functor function pairs are inferred from that listing; i.e. Diagram D in Figure 3-5 is defined as follows:

$$\text{Objects}(D) = \{A, B, C, D\}$$
$$\text{Arrows}(D) = \{ f, g, h, i, \ id_A, id_B, id_C, id_D \}$$

This latter definition of a diagram does not normally allow the same object (or arrow) to occur in a diagram multiple times, as the notion of a set eliminates duplicate items. However, it is assumed that once a diagram is defined, or given, each object and arrow in that diagram is indexed so as to distinguish it. This indexing enables a diagram to be taken apart and reformed without equating what would ordinarily be identical objects and arrows.

3.3.2 Diagram morphisms

The definition of a morphism between diagrams is based on a "content preserving" arrow category diagram and a "structure-preserving" shape morphism. Definition 3.12 and Figure 3-10 are based on the definition of diagram refinement as presented for diagrams in category *Spec* in [SJ95], where the arrows between diagrams are collections of compatible interpretations between specifications.

Definition 3.12. Given that $D_1:G_1 \rightarrow X$ and $D_2:G_2 \rightarrow X$ are arbitrary diagrams in category X (i.e. arbitrary functors form arbitrary shape categories G_1 and G_2 to category X) then a *morphism between diagrams* is a pair of functors $<\delta, \sigma>$, where $\delta:G_1 \rightarrow X^{\rightarrow}$ is a diagram in category $X^{\rightarrow}$ and $\sigma:G_1 \rightarrow G_2$ is a shape functor, such that the *Cat*-diagram (a diagram in the category of categories and the functors between categories) in Figure 3-10 commutes.

$$\begin{tikzcd}
 & G_1 \arrow[rr, "\sigma"] \arrow[dl, "D_1"] \arrow[d, "\delta"] \arrow[dr] & & G_2 \arrow[dl, "D_2"] \\
X & X^{\rightarrow} \arrow[l, "\mathit{Source}"'] \arrow[r, "\mathit{Target}"] & X &
\end{tikzcd}$$

Figure 3-10. The defining diagram for a morphism between diagrams

Diagram D_1 in Figure 3-10 is a functor that assigns the nodes and arcs of a shape category, G_1, to the objects and arrows of the target category X. Diagram D_2 is similarly defined over a shape category G_2. Functor σ represents the shape morphism between shapes G_1 and G_2 in the category of shapes that ensures that the shape of G_2 is compatible with shape G_1. For example, σ could be the identity functor ($G_1 = G_2$), σ could be a monic shape embedding where shape G_1 is wholly embedded in shape G_2 and shape G_2 may contain additional nodes and arcs, or σ could be epic in which case part of G_1's shape may be "collapsed" in G_2. Thus, via the top and side arrows in Figure 3-10, the shape of the category X diagram D_1 is compatible with that of the category X diagram D_2.

The functor (diagram) δ assigns the nodes and arcs of shape category G_1 to a collection of $X^{\rightarrow}$ objects and arrows. In this case the objects of δ are the needed X-arrows between the category X diagrams D_1 and D_2, as category $X^{\rightarrow}$-objects are category X-arrows. The arrows of diagram δ are pairs of arrows in category X. The first arrow of the pair is in diagram D_1 and the second arrow is in diagram D_2 such that there is a commuting square between diagrams D_1 and D_2. For the diagram in Figure 3-10 to commute, functor (diagram) D_1 must be equal to functor (diagram) *Source* $\circ$ δ, and functor (diagram) $D_2 \circ \sigma$ must be equal to functor (diagram) *Target* $\circ$ δ. This latter diagram has the objects and arrows of diagram D_2 but the shape of diagram D_1 as indicated by the functor composition $D_2 \circ \sigma$. The objects and the arrows of diagram D_1 are preserved in diagram D_2, because the entire diagram in Figure 3-10 must commute for the pair $<\delta, \sigma>$ to be considered a valid diagram morphism the structure.

As an example of a diagram morphism, the morphism between diagrams from Figure 3-1 is depicted in greater detail in Figure 3-11, where the underlying structure is revealed. While the functors D_1,

D_2, and σ are faithfully rendered in Figure 3-11, the functors δ, *Source*, and *Target* (arrows $X \leftarrow X^{\rightarrow} \rightarrow X$) are not because the arrows between the X diagrams are depicted as X-arrows instead of as a diagram of $X^{\rightarrow}$-arrows.

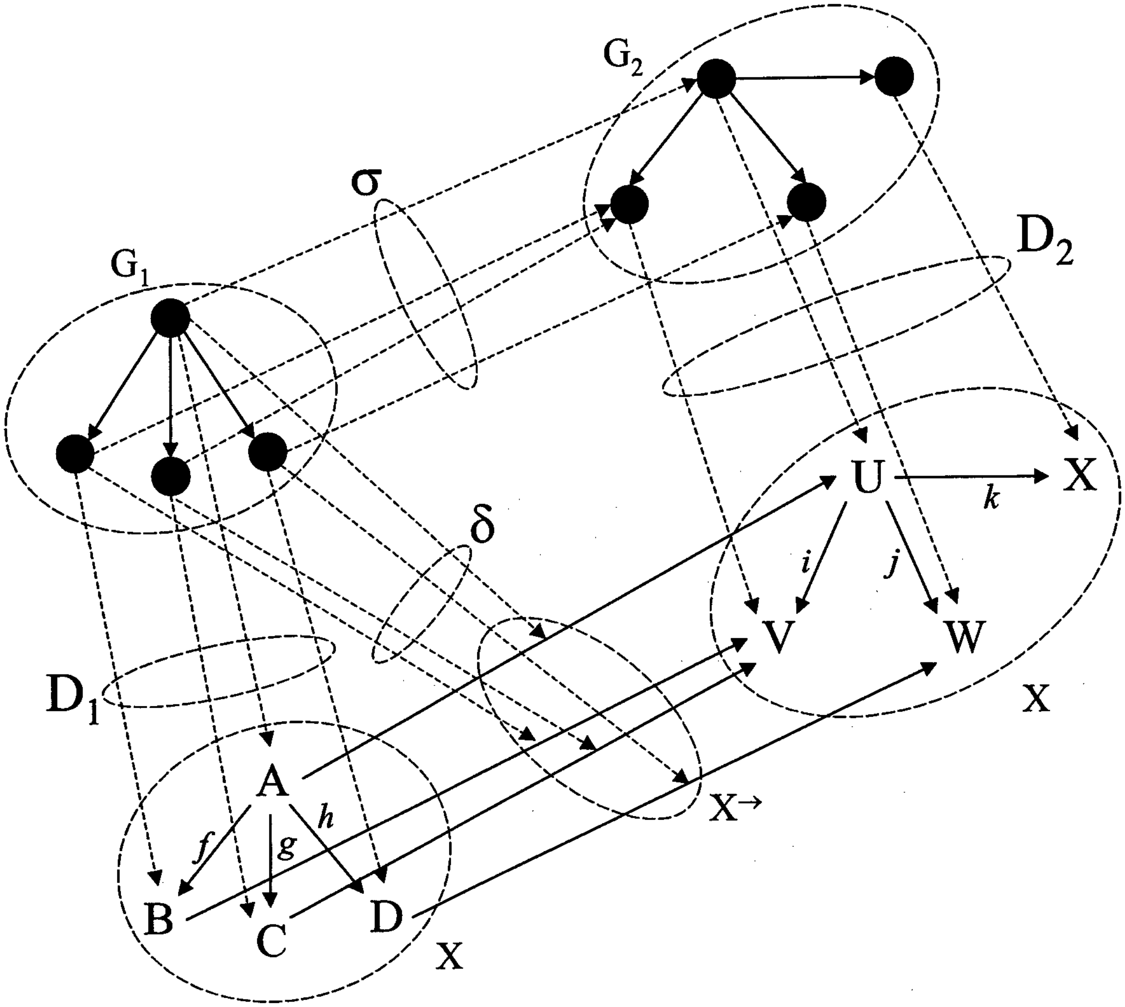


Figure 3-11. Diagram Morphism < δ, σ >:$D_1 \rightarrow D_2$

In Figure 3-12 the functors δ, *Source*, and *Target* are accurately portrayed. The restriction of diagram D_2, $D_2 \circ \sigma$, commutes with the functor *Target* • δ. The arrows that connecting diagram D_1 to diagram $D_2 \circ \sigma$ in Figure 3-12 are X-arrows, the same arrows as those that connected diagram D_1 to diagram D_2 in Figure 3-11.

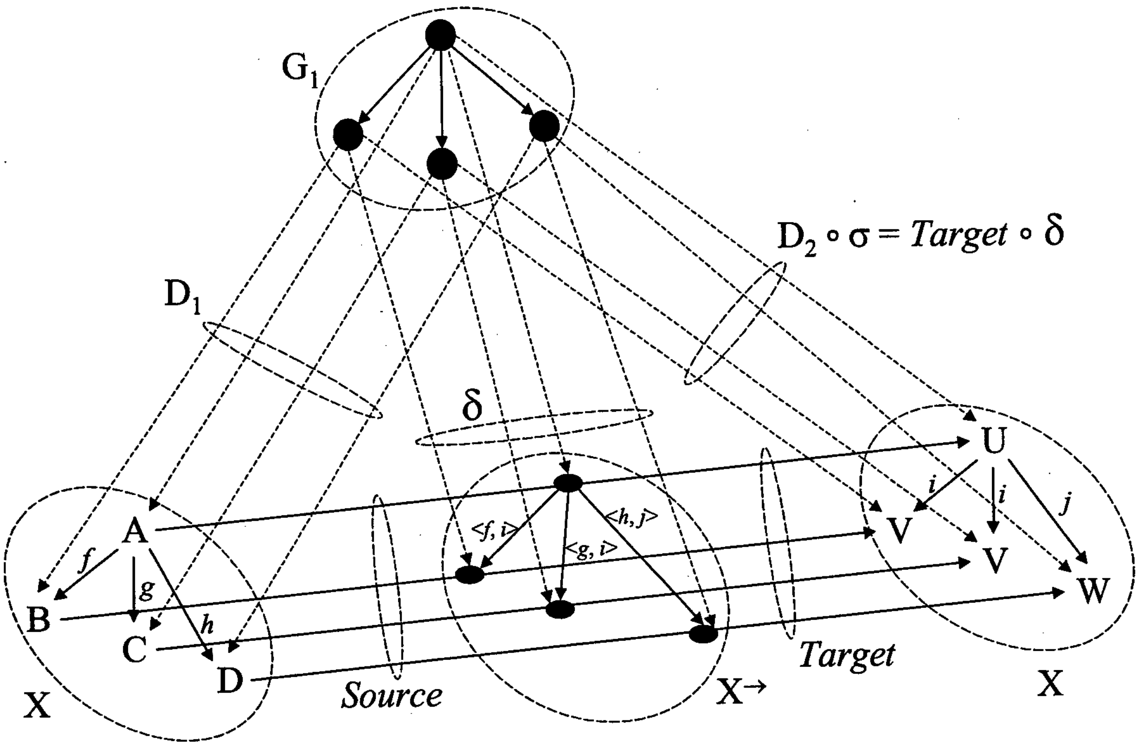


Figure 3-12. Functor δ:$G_1 \rightarrow X^{\rightarrow}$

3.3.3 The Category d*X*

The diagrams of a particular category, and the morphisms between those diagrams, are themselves the objects and arrows of a category.

Proposition 3.13. Given a category X, there exists a category of diagrams and diagram morphisms of category X called category *d*X. The category *d*X objects are all the diagrams of category X. The category *d*X arrows are all the diagram morphisms of the category X diagrams.

Proof: Category *d*X-arrows compose and are associative because (collections) of category X arrows compose and are associative and shape morphisms compose and are associative.

Identity Arrow: The identity arrow, id_D, for a given *d*X-object D:G→X, is the pair of functors $<id_\delta, id_G>$, where id_G is the identity shape morphism for shape category G, and where diagram id_δ:$G \rightarrow X^{\rightarrow}$ is the following pair of functions.

$\delta_{Object}(\alpha) = \{ id_{D(\alpha)} \mid \alpha \in \text{Objects}(G) \}$, i.e. identity arrows of the X-objects in D,
$\delta_{Arrow}(f) = \{ < D(f), D(f) > \mid f \in \text{Arrows}(G) \}$.

The identity *d*X-arrow <δ, σ> defined above is the left and right identity for the composition operation (see below) as σ is a shape category identity functor and the nodes in diagram δ that connect diagram D with itself are the identity arrows of the X-objects making up the diagram.

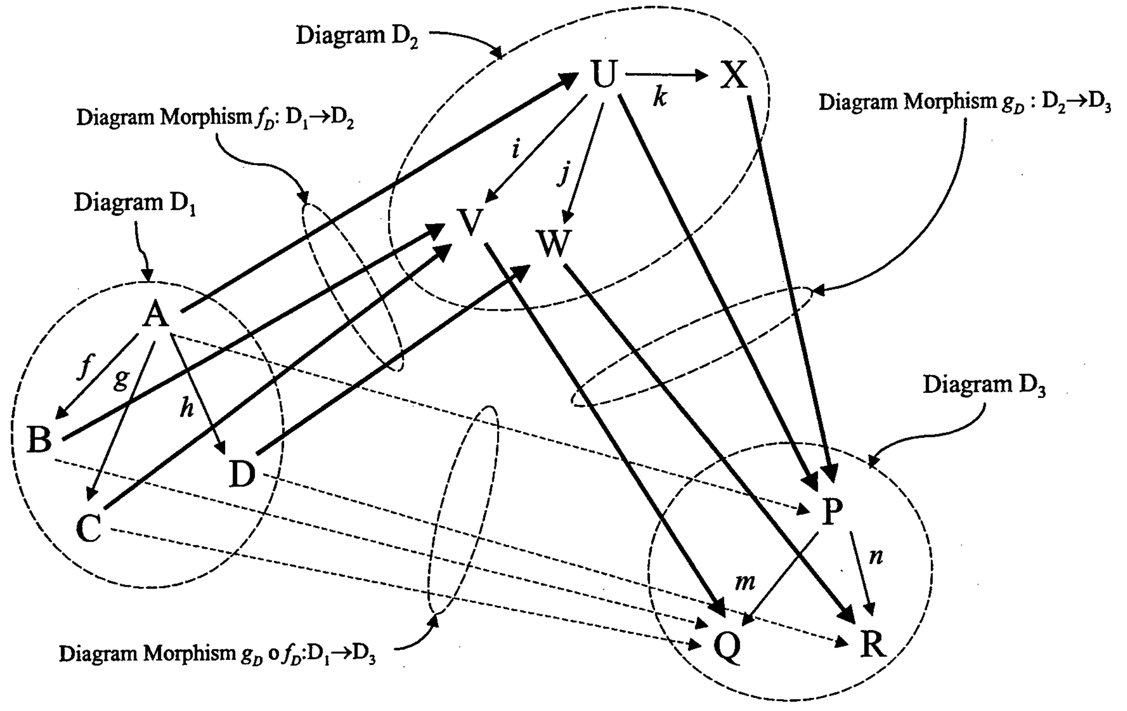


Diagram D_2
Diagram Morphism f_D: $D_1 \rightarrow D_2$
Diagram Morphism g_D : $D_2 \rightarrow D_3$
Diagram D_1
U
X
k
i
j
V
W
A
f
g
h
B
C
D
Diagram D_3
P
n
m
Q
R
Diagram Morphism $g_D \circ f_D$:$D_1 \rightarrow D_3$

Figure 3-13. Example diagram morphism composition

Composition of arrows: Successive morphisms between diagrams compose (as depicted in Figure 3-13) because the juxtaposition of two Figure 3-10 diagrams representing the composition of two successive diagram morphisms, $<\delta_1, \sigma_1> \circ <\delta_2, \sigma_2>$, "compose" as depicted in Figure 3-14.

$$\begin{tikzcd}
 & G_1 \arrow[rr, "\sigma_1"] \arrow[dl, "D_1"'] \arrow[d, "\delta_1"] \arrow[dr] & & G_2 \arrow[rr, "\sigma_2"] \arrow[dl, "D_2"'] \arrow[d, "\delta_2"] \arrow[dr] & & G_3 \arrow[dl, "D_3"] \\
X & X \arrow[l, "Source"'] \arrow[r, "Target"'] & X & X \arrow[l, "Source"'] \arrow[r, "Target"'] & X &
\end{tikzcd}
\longrightarrow
\begin{tikzcd}
 & G_1 \arrow[rr, "\sigma_2 \circ \sigma_1"] \arrow[dl, "D_1"'] \arrow[d, "(\delta_2 \circ \sigma_1) \bullet \delta_1"] \arrow[dr] & & G_3 \arrow[dl, "D_3"] \\
X & X \arrow[l, "Source"'] \arrow[r, "Target"'] & X &
\end{tikzcd}$$

Figure 3-14. Composition of diagram morphisms

In Figure 3-14, functors σ_1 and σ_2 compose to form a single shape morphism, $\sigma_2 \circ \sigma_1$, from shape G_1 to shape G_3. Functor $\delta_2 \circ \sigma_1$ restricts the shape of functor δ_2 to that of shape G_1. (If shape G_2 has additional arcs and nodes not "covered" by G_1 then these extra nodes and arcs are removed. If some of G_1's arcs and nodes are "collapsed" in their mapping to shape G_2 then these arcs and nodes are "expanded" in the diagram.) For an example of a shape restriction, compare functor (diagram) D_2 in Figure 3-11 with functor (diagram) $D_2 \circ \sigma$ in Figure 3-12. Functors $\delta_2 \circ \sigma_1$ and δ_1, which are category $X^{\rightarrow}$ diagrams, are then composed using a variation of the parallel composition operation of category $X^{\rightarrow}$.

The parallel composition operation • in $X^{\rightarrow}$ (Definition 3.10) can be "raised" to an operation on diagrams in $X^{\rightarrow}$. Given diagram functors δ_A:G→$X^{\rightarrow}$ and δ_B:G→$X^{\rightarrow}$, where *Source*(δ_B) = *Target*(δ_A), the "raised" • operation composes δ_A and δ_B to form a functor δ_C:C→$X^{\rightarrow}$ where the objects in δ_C are the composition of the adjacent objects (X-arrows) in δ_A and δ_B and where *Source*(δ_C) = *Source*(δ_A) and *Target*(δ_C) = *Target*(δ_B). (Given a commuting diagram as depicted with solid arrows in Figure 3-15, the dashed arrows are formed by the • operator and are also commuting arrows with the rest of the diagram.)

$$\begin{tikzcd}
 & & G \arrow[lld, "D_1"'] \arrow[ld, "\delta_A"'] \arrow[d, "D_2"] \arrow[rd, "\delta_B"] \arrow[rrd, "D_3"] \arrow[dd, dashed, bend right, "\delta_C = \delta_B \bullet \delta_A"] & & \\
X & X^{\rightarrow} \arrow[l, "\textit{Source}"'] \arrow[r, "\textit{Target}"] & X & X^{\rightarrow} \arrow[l, "\textit{Source}"'] \arrow[r, "\textit{Target}"] & X \\
 & & X^{\rightarrow} \arrow[llu, dashed, "\textit{Source}"] \arrow[rru, dashed, "\textit{Target}"'] & &
\end{tikzcd}$$

Figure 3-15. Parallel composition operation over diagrams in category $X^{\rightarrow}$

An example of this is illustrated in Figure 3-16, which depicts one such composition from the example diagram morphism of Figure 3-13. The • operation over category $X^{\rightarrow}$ diagrams is merely the union of the • operations over the individual adjacent commuting squares making up the $X^{\rightarrow}$ diagrams.

Because functors δ_1 and ($\delta_2 \circ \sigma_1$) in Figure 3-14 have the same shape category, G_1, as the domain category and they have a common "codomain/domain", (Cod $\circ\ \delta_1$) = (Dom $\circ\ \delta_2 \circ \sigma_1$), the objects (X-arrows) making up functors δ_1 and ($\delta_2 \circ \sigma_1$) can be individually composed. That composition also has shape G_1 and is expressed as ($\delta_2 \circ \sigma_1$) • δ_1, i.e. a shape restriction ($\delta_2 \circ \sigma_1$) followed by the composition of a collection of compatible commutative squares (composing a collection of pairs of arrows in $X^{\rightarrow}$ with the • operation). Thus $< \delta_1, \sigma_1 > \circ < \delta_2, \sigma_2 > = < (\delta_2 \circ \sigma_1) \bullet \delta_1,\ \sigma_2 \circ \sigma_1 >$.

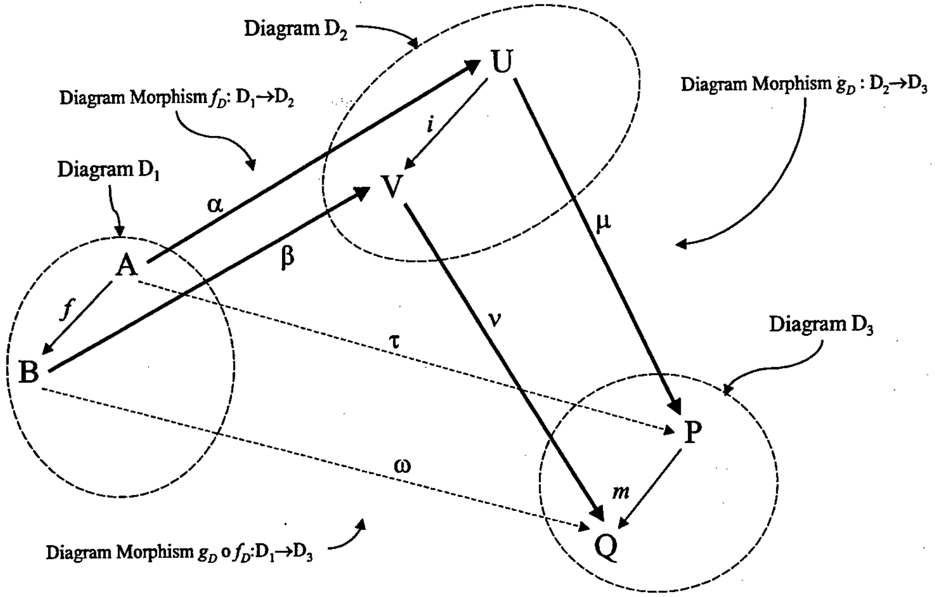


Diagram D$_2$
U
Diagram Morphism f_D: D$_1$→D$_2$
Diagram Morphism g_D : D$_2$→D$_3$
i
Diagram D$_1$
V
α
μ
A
β
f
ν
Diagram D$_3$
τ
B
P
ω
m
Q
Diagram Morphism $g_D \circ f_D$:D$_1$→D$_3$

Figure 3-16. Parallel composition operation in context

Associativity of arrows: The proof of the associativity of diagram morphisms relies on the juxtaposition of three Figure 3-10 diagrams as depicted in Figure 3-17.

$$\begin{tikzcd}
& G_1 \arrow[rr, "\sigma_1"] \arrow[dl, "D_1"] \arrow[d, "\delta_1"] \arrow[drr] & & G_2 \arrow[rr, "\sigma_2"] \arrow[dl, "D_2"] \arrow[d, "\delta_2"] \arrow[drr] & & G_3 \arrow[rr, "\sigma_3"] \arrow[dl, "D_3"] \arrow[d, "\delta_3"] \arrow[drr] & & G_4 \arrow[dl, "D_4"] \\
X & X \arrow[l, "\text{Source}"] \arrow[r, "\text{Target}"] & X & X \arrow[l, "\text{Source}"] \arrow[r, "\text{Target}"] & X & X \arrow[l, "\text{Source}"] \arrow[r, "\text{Target}"] & X &
\end{tikzcd}$$

Figure 3-17. Associativity of diagram morphisms

The proof that the composition of diagram morphisms is associative,

i.e. $(<\delta_1, \sigma_1> \circ <\delta_2, \sigma_2>) \circ <\delta_3, \sigma_3> \; = \; <\delta_1, \sigma_1> \circ (<\delta_2, \sigma_2> \circ <\delta_3, \sigma_3>)$, relies on the associativity of functors, the associativity of the • operation, and the factoring of a shape "restriction" over a • operation.

$(< \delta_1, \sigma_1 > \circ < \delta_2, \sigma_2 >) \circ < \delta_3, \sigma_3 >$	Given: Left associative equation
$< (\delta_2 \circ \sigma_1) \bullet \delta_1,\ \sigma_2 \circ \sigma_1 > \circ < \delta_3, \sigma_3 >$	First composition of arrows in *d*X
$< (\delta_3 \circ (\sigma_2 \circ \sigma_1)) \bullet ((\delta_2 \circ \sigma_1) \bullet \delta_1),\ \sigma_3 \circ \sigma_2 \circ \sigma_1 >$	Second composition of arrows in *d*X
$< ((\delta_3 \circ \sigma_2) \circ \sigma_1) \bullet ((\delta_2 \circ \sigma_1) \bullet \delta_1),\ \sigma_3 \circ \sigma_2 \circ \sigma_1 >$	Associativity of functor composition, ∘
$< (((\delta_3 \circ \sigma_2) \circ \sigma_1) \bullet (\delta_2 \circ \sigma_1)) \bullet \delta_1,\ \sigma_3 \circ \sigma_2 \circ \sigma_1 >$	Associativity of parallel composition, •
$< ((((\delta_3 \circ \sigma_2) \bullet \delta_2) \circ \sigma_1) \bullet \delta_1),\ \sigma_3 \circ \sigma_2 \circ \sigma_1 >$	Factoring out the shape restriction, σ_1
$< \delta_1, \sigma_1 > \circ (< (\delta_3 \circ \sigma_2) \bullet \delta_2,\ \sigma_3 \circ \sigma_2 >)$	First (un)composition of arrows in *d*X
$< \delta_1, \sigma_1 > \circ (< \delta_2, \sigma_2 > \circ < \delta_3, \sigma_3 >)$	Second (un)composition of arrows in *d*X

There exists a diagram category *d*X for each category X because diagram morphisms between diagrams of a category X compose and are associative and there exists an identity diagram morphism for each X diagram. □

3.4 *Partitioning and Flattening Diagrams*

In order to be able to operate freely over category X and category *d*X, the relationships that exist between the two categories must be formalized. The relationships defined in this section are used in Chapter 6 to describe the representation of diagram information.

3.4.1 The functor *Diagramize*

The functor *Diagramize*:X→*d*X takes X-objects to "singleton" X diagrams and takes X-arrows to diagram morphisms between the singleton diagrams.

Definition 3.14. *Diagramize* is the pair of functions < $Diagramize_{\text{Object}}$, $Diagramize_{\text{Arrow}}$ > from X-objects and X-arrows respectively to *d*X-objects and *d*X-arrows. $Diagramize_{\text{Object}}$ and $Diagramize_{\text{Arrow}}$ are defined as follows:

For each $\alpha \in$ Objects(X), $Diagramize_{\text{Object}}(\alpha)$ = D:G→X,
where G = ●↺, D(●) = α, and D(↺) = id_{α}.

For each $f \in$ Arrows(X), $Diagramize_{\text{Arrow}}(f)$ = <δ:G→$X^{\rightarrow}$, σ>,
where σ = ●↺→●↺, G = ●↺, δ(●) = *f*, δ(↺) = <$id_{\text{dom}(f)}$, $id_{\text{cod}(f)}$>.

Proposition 3.15. *Diagramize* is a functor from category X to category *d*X.

Proof: *Diagramize* preserves identity arrows as a category X identity arrow $A \xrightarrow{id} A$ (i.e. id_A) becomes the diagram morphism identity arrow, <δ, ●↺ → ●↺ >, where δ(●) = id_A and δ(↺) = {< id_A, id_A >}. *Diagramize* preserves arrow compositions as all of the resulting diagram morphisms have the same shape, σ = ●↺→●↺, thus the shape morphism component of a diagram composes. Also the $X^{\rightarrow}$-objects of diagram δ, being parallel X-arrows, compose under the parallel composition operation, •, i.e.

Diagramize(A→B $\circ_X$ B→C) = *Diagramize*(A→B) • *Diagramize*(B→C) □

3.4.2 The functor *Colimit*

If category X is cocomplete then there exists a functor, *Colimit*:*d*X→X, from category *d*X to category X based on the colimit operation on diagrams in category X.

Definition 3.16. *Colimit* is a pair of functions < $Colimit_{\text{Object}}$, $Colimit_{\text{Arrow}}$ > from category *d*X objects and arrows respectively to category X objects and arrows. $Colimit_{\text{Object}}$ and $Colimit_{\text{Arrow}}$ are defined as follows (under the assumption that category X is cocomplete):

For each $\alpha \in$ Objects(dX), $Colimit_{\text{Object}}(\alpha) = \text{colimit}_X(\alpha)$,
where colimit_X is the colimit operation of category X.

For each $f \in$ Arrows(dX), $Colimit_{\text{Arrow}}(f)$ is the universal X-arrow from the X-object $Colimit_{\text{Object}}(\text{dom}(f))$ to the X-object $Colimit_{\text{Object}}(\text{cod}(f))$, as the latter object has a cocone to it from the diagram dom(f) (as depicted in Figure 3-18).

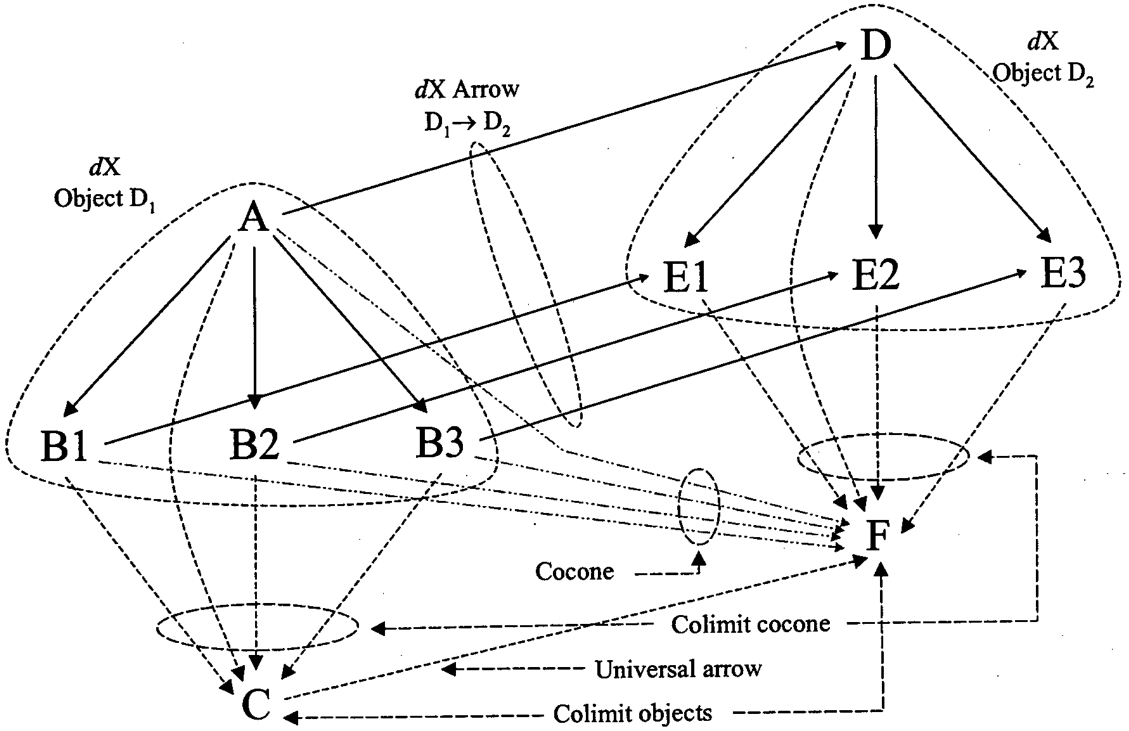


Figure 3-18. Existence and uniqueness of an X arrow induced by the *Colimit*:*d*X→X functor

Proposition 3.17. *Colimit* is a functor from category *d*X to category X.

Proof: Identity arrows are preserved by *Colimit* as the universal arrow from an object to itself is the identity arrow. Arrow composition is also preserved as two successive diagram morphisms $D_1 \rightarrow D_2 \rightarrow D_3$ along with their colimit objects can be viewed as a large category X diagram, and in that category the cocone morphisms and the universal arrows between the colimit objects are all commuting arrows. □

Functors *Diagramize* and *Colimit* have an interesting relationship when viewed in conjunction with diagrams in category X and diagrams in category *d*X. The two functors enable diagrams in one category to be viewed as diagrams in the other category. Figure 3-19 depicts a specific example of a category X diagram and a category *d*X diagram related by functors *Diagramize* and *Colimit*. The two functors are not inverses as *Diagramize* is a monic functor whereas *Colimit* is not, i.e. there may be many category *d*X diagrams that the *Colimit* functor maps to the same elements within category X. Note that in

Figure 3-19, functor D is a category X diagram and functor $\mathcal{D}$ is a category *d*X diagram and that domain(D) = domain($\mathcal{D}$).

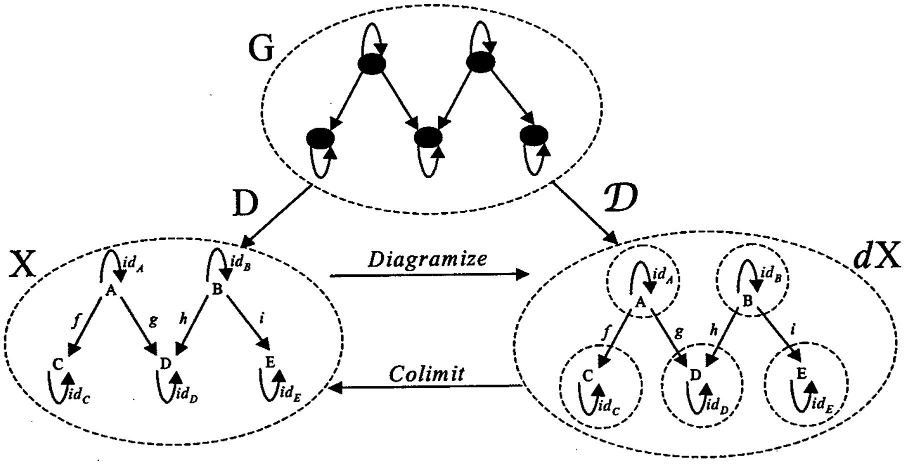
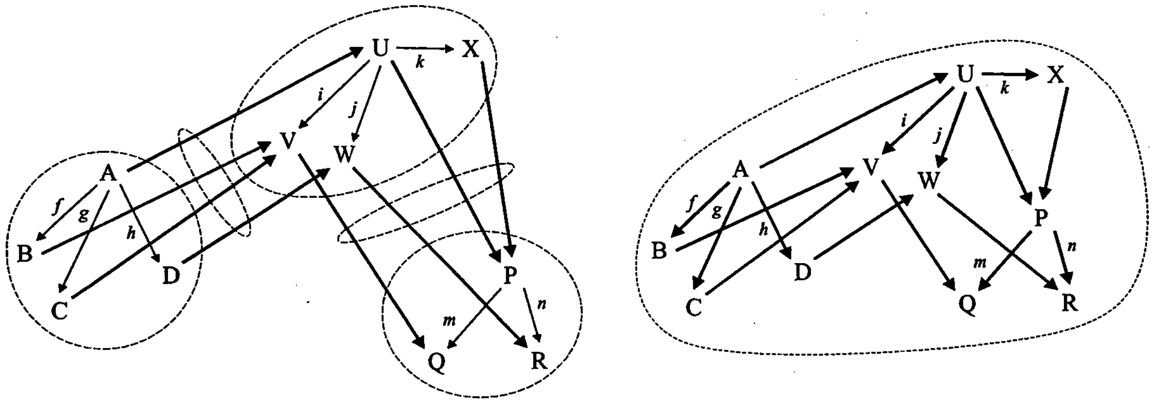


Figure 3-19. Relationship between functors *Diagramize* and *Colimit*

3.4.3 Flattening a category *d*X diagram

Any category *d*X diagram can be viewed as a category X diagram by removing the additional structure associated with the category *d*X objects and arrows and adding in the additional arrows and arrow compositions induced by the diagram morphisms. Thus, for example, the category *d*X diagram $D_1 \rightarrow D_2 \rightarrow D_3$ from Figure 3-13 can be viewed as the category X diagram as depicted in Figure 3-20. Note that Figure 3-20 does not depict any identity arrows or arrow compositions in order to keep the diagram simple. The function *Flatten* is defined to return a category X diagram based on the category X objects and arrows underlying the diagrams and diagram morphisms in a given category *d*X diagram.



Category *d*X diagram $\mathcal{D}$

Diagram $\mathcal{D}$ flattened so that it is a category X diagram

Figure 3-20. Flattening a category *d*X diagram

Definition 3.18. *Flatten* is a function from category *d*X diagrams to category X diagrams defined as follows:

Flatten($\mathcal{D}$) = $D{:}G \to X$

Let $\mathcal{D}{:}G_0 \to dX$ be a *d*X diagram,
Let $D_i{:}G_i \to X$ be *d*X objects of $\mathcal{D}$, i.e. $D_i = \mathcal{D}(\alpha)$ for some $\alpha \in$ Objects(G_0)
Let $< \delta_j, \sigma_j >$ be *d*X arrows of $\mathcal{D}$, i.e. $< \delta_j ,\sigma_j > = \mathcal{D}(f)$ for some $f \in$ Arrows(G_0)

Shape G is defined as follows:

Objects(G) = $\bigcup$ Objects(G_i),
i.e. a union of the nodes associated with each shape G_i of a diagram D_i of $\mathcal{D}$.

Arrows(G) = $\bigcup$ Arrows(G_i) $\cup$ $\{\alpha \to \sigma_j(\alpha) \mid \alpha \in$ Objects(dom(σ_j))$\}$
and closed under composition
i.e. a union of the arrows associated with each shape G_i of a diagram D_i of $\mathcal{D}$ unioned with arrows between the nodes of different diagrams constructed based on the shape morphisms (σ_j) between the diagrams of $\mathcal{D}$, and then the result closed under composition.

Functor $D{:}G \to X$ is the pair of functions, $< D_{Object}, D_{Arrow} >$ defined as follows:

For each $\alpha \in$ Objects(G), $D_{Object}(\alpha) = D_i(\alpha)$ for an α that originated in G_i

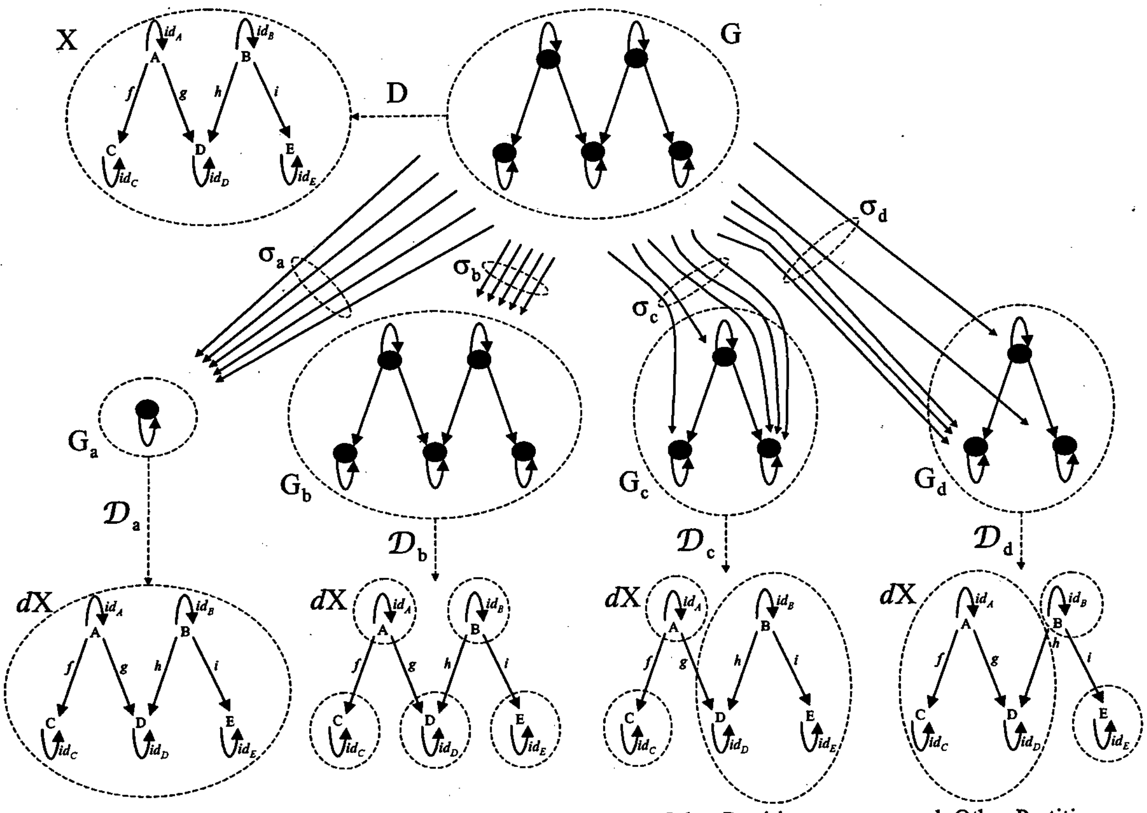
For each $f \in$ Arrows(G)
$D_{Arrow}(f) = D_i(f)$ for an f that originated from G_i
$\delta_i(\text{dom}(f))$ for an f that originated as an arrow between the G_i shapes
$D_{Arrow}(f_1) \circ D_{Arrow}(f_2)$ for an f that originated as the composition $f_1 \circ f_2$

3.4.4 Partitions of a category X diagram

Informally, the act of partitioning induces a category *d*X diagram made up of the objects and arrows of a given category X diagram. Flattening the resulting partition (the category *d*X diagram) must result in the original category X diagram from which it was derived.

At the top of Figure 3-21 is a category X diagram, $D{:}G \to X$. This category X diagram is partitioned into four different category *d*X diagrams. The shapes G_a through G_d and the morphisms to them from shape G indicate how the category X diagram D can be partitioned into category *d*X diagrams. In Figure 3-21(a), diagram D is partitioned as a unit into a singleton *d*X diagram $\mathcal{D}_a$. This singleton *d*X diagram is called the trivial partition of the X diagram. In Figure 3-21(b), each individual specification in diagram D is viewed as a *d*X object and the arrows in diagram D become diagram morphisms between those *d*X objects as depicted in the *d*X diagram $\mathcal{D}_b$. This is called the maximal partition of the X diagram and this maximal partition can also be constructed using the functor *Diagramize* defined in Section 3.4.1.

Every X diagram can be interpreted as a *d*X diagram using either a trivial partition or a maximal partition of the X diagram. If the X diagram is itself a singleton then the trivial and maximal partitions in *d*X are identical. There may be other ways to partition any given X diagram into a *d*X diagram such as depicted in Figure 3-21(c) and Figure 3-21(d) for the given diagram D. Arbitrary "partitions" of an X diagram into collections of objects and arrows may not result in a *d*X diagram though as it may not be possible to form a diagram morphism between the partition elements.



a. Trivial Partition b. Maximal Partition c. Other Partition d. Other Partition

Figure 3-21. Partitioning a category X diagram into different category *d*X diagrams

Informally, a partition morphism (such as the shape morphisms σ_a, σ_b, σ_c and σ_d in Figure 3-21) is used to break up a category X diagram into sub-diagrams such that well formed diagram morphisms exist between the sub-diagrams. A partition morphism is a shape epimorphism with additional properties related to the arrows between the objects making up the partition elements. A partition morphism indicates which category *d*X diagram to derive from a category X diagram, the trivial partition, the maximal partition, or some other partition. Given a diagram D:G→X, the trivial partition, Figure 3-21(a), is formed by the partition morphism G→●⟲, which indicates that all of the objects and arrows in of diagram D should form a single *d*X diagram. The maximal partition, Figure 3-21(b), is formed by the partition morphism $\sigma_b = id_G$ (the identity shape morphism), which indicates that each individual X object of diagram D should form its own *d*X diagram.

Definition 3.19. A *Partition morphism* is an epimorphism between two shapes, $\sigma:G_1 \rightarrow G_2$, that satisfies the following four predicates:

```
1)- Each collapsed source node has an arrow to a collapsed target node
    For each non-identity arrow $f \in$ Arrows($G_2$),
        Let Collapsed-source-nodes = $\{\alpha \mid \alpha \in$ Objects($G_1$) and $\sigma(\alpha) =$ dom($f$) $\}$.
        For each $\alpha \in$ Collapsed-source-nodes
            there must exist a $\alpha \rightarrow \beta \in$ Arrows($G_1$)
            such that $\sigma(\alpha \rightarrow \beta) = f$.

2)- Each collapsed source arrow has an associated collapsed target arrow
    For each non-identity arrow $f \in$ Arrows($G_2$),
        Let Collapsed-source-arrows = $\{g \mid g \in$ Arrows($G_1$) and $\sigma(g) = id_{\mathrm{dom}(f)}$
        For each $g \in$ Collapsed-source-arrows
            there must exist a $\alpha \rightarrow \beta \in$ Arrows($G_1$)
            such that $\sigma(\alpha \rightarrow \beta) = id_{\mathrm{cod}(f)}$
            and there exist arrows dom($g$)$\rightarrow\alpha$, cod($g$)$\rightarrow\beta \in$ Arrows($G_1$)

3)- Collapsed arrows with the same domain node must have distinct codomain nodes
    For each non-identity arrow $f \in$ Arrows($G_2$),
        for each distinct $h, i \in$ Arrows($G_1$),
        where $\sigma(h) = f \wedge \sigma(i) = f \wedge$ dom($h$) = dom($i$),
        it must be the case that cod($h$) $\neq$ cod($i$).

4)- Collapsed arrows with the same domain node must have connected codomain nodes
    For each non-identity arrow $f \in$ Arrows($G_2$),
        for each distinct $h, i \in$ Arrows($G_1$),
        where $\sigma(h) = f \wedge \sigma(i) = f \wedge$ dom($h$) = dom($i$),
        it must be the case that there exists an arrow $g$:cod($h$)$\rightarrow$cod($i$) $\in$ Arrows($G_1$) or
                                              an arrow $g$:cod($i$)$\rightarrow$cod($h$) $\in$ Arrows($G_1$)
```

For a given diagram $D:G_1 \rightarrow X$ and a partition morphism $\sigma:G_1 \rightarrow G_2$, the four predicates in Definition 3.19 are necessary to ensure that the objects and arrows in G_2 induce a *d*X diagram that is based on diagram D. These predicates are necessary to indicate a partition but not sufficient as the commutative nature of a diagram cannot be stated using only the shape of a diagram.

Figure 3-22 depicts examples of shape epimorphisms that are partition morphisms. Each partition morphism $\sigma:G_1 \rightarrow G_2$ in Figure 3-22 can be used to induce a diagram, G_3, in the category of shapes. The induced diagram G_3 has shape G_2 but its "contents" are based on shape G_1. Essentially, any category X diagram $D:G_1 \rightarrow X$ (where G_1 is any of the shapes in Figure 3-22) can be partitioned into a *d*X diagram with shape G_2 and whose contents are reflected by the mapping from the shape G_1 contents to the shape G_3 contents.

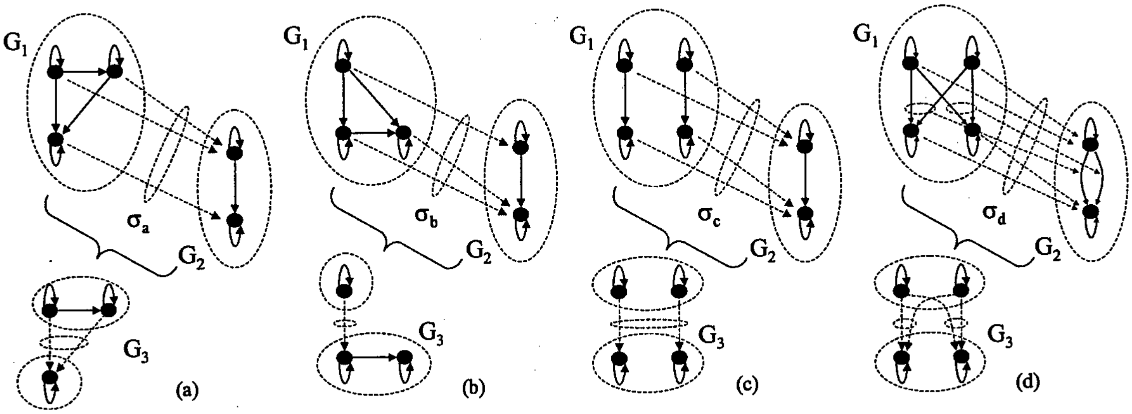


Figure 3-22. Shape epimorphisms that are partition morphisms

In contrast to Figure 3-22, each shape epimorphism in Figure 3-23 violates one or more of the predicates in Definition 3.19 and therefore is not partition morphism. In each case the violation indicates that a diagram G_3 in the category of shapes cannot be induced because either there does not exist an arrow that can be used for a diagram morphism or there exists too many arrows to choose from and the choices are mutually exclusive.

For example, shape morphisms σ_a in Figure 3-23 violates predicate (1). A required arrow for a diagram morphism does not exist in shape G_1, and therefore the shape of a diagram morphism cannot be induced by σ_a, (compare Figure 3-22(a) with Figure 3-23(a)). Shape morphism σ_b, Figure 3-23(b), violates predicate (3). Only one of the two arrows can be used for the shape of the diagram morphism and the choice is mutually exclusive (assuming that the underlying arrows are not the same arrow). Shape morphisms σ_c and σ_d violate predicate (4). More than one arrow choice exists for a diagram morphism and the choices are mutually exclusive. Shape morphism σ_e violates predicates (1) and (4). Shape morphism σ_f violates predicate (2) as there does not exist a target arrow associated with the indicated source arrow and therefore a commuting square in the underlying category could never be formed. A diagram $D:G_1\rightarrow X$ with any shape G_1 in Figure 3-23 cannot use the associated shape morphism to induce a *d*X diagram shape. In every case either the required diagram morphism cannot be formed, or if formed with an arbitrary choice some of the structure from the category X diagram will be lost.

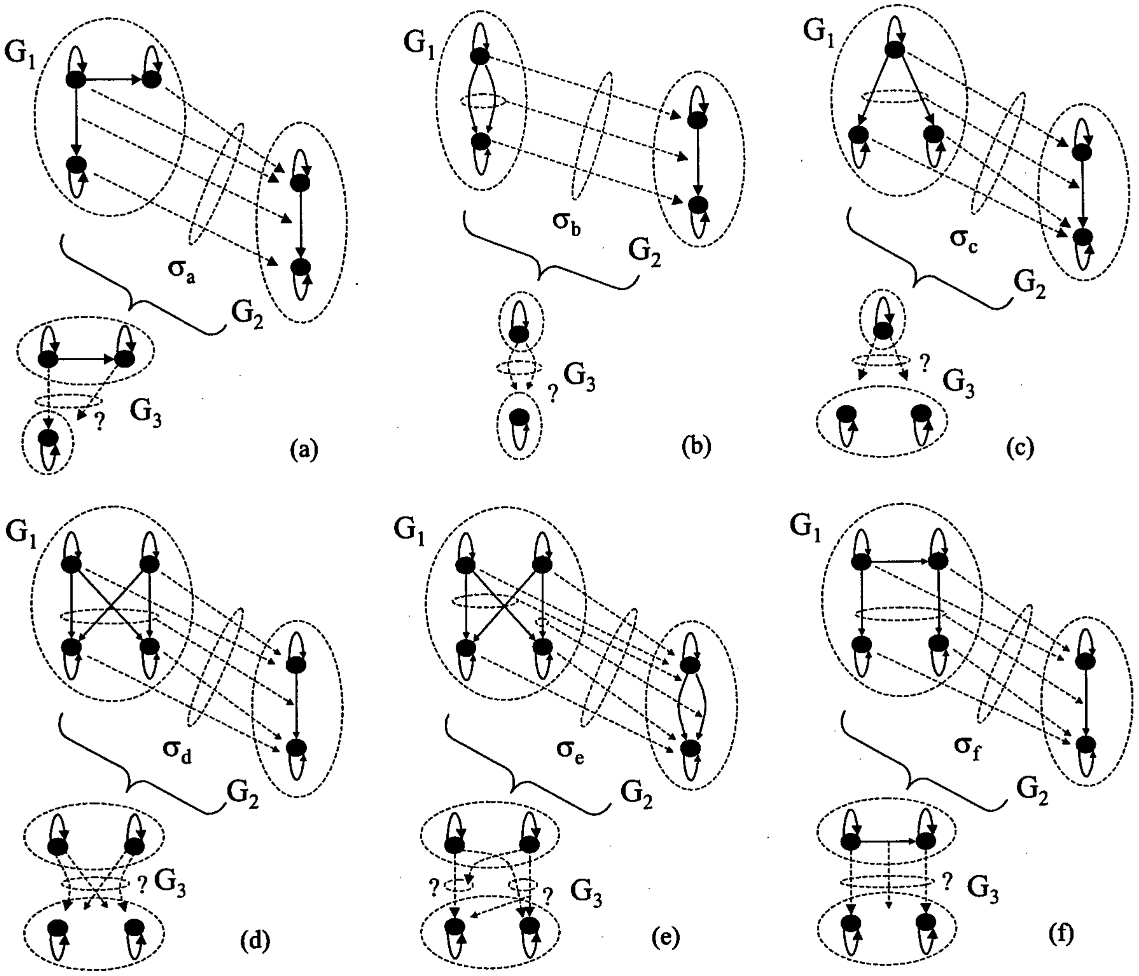


Figure 3-23. Shape epimorphisms that are *not* partition morphisms

Note that shape morphism σ_b in Figure 3-22(b) also appears to have more than one choice for the diagram morphism in shape G_3. However, because the codomain nodes involved in the choice are connected, i.e. predicate (4), the arrow choice that was not taken can be derived, by arrow composition, from the arrow choice that was taken. In Figure 3-22(b), even though one of the arrows was not used, it can be recovered. For the case depicted in Figure 3-23(c), deriving the arrow choice that was not taken is not possible as predicate (4) is violated (the target nodes are not connected via an arrow) and therefore arrow composition cannot be used to recover the arrow choice that was not taken.

Definition 3.20. The (partial) function *Partition* takes a category X diagram $D:G\rightarrow X$ and a partition morphism $\sigma:G\rightarrow G_0$ and returns a category *d*X diagram, $\mathcal{D}:G_0\rightarrow dX$, called a *partition* of diagram D defined as follows:

$$\text{Partition}(D, \sigma) = \mathcal{D} \text{ iff Flatten}(\mathcal{D}) = D.$$

The function partition is partial as it may not be defined if D is not a commuting diagram. As an example, in Figure 3-24, if diagram D:G₁→X is a commuting diagram then diagram 𝒟:G₃→*d*X is defined. If diagram D is not a commuting diagram then diagram 𝒟 is not defined as the X-arrows making up the diagram morphism 𝒟 cannot form a commuting square. Not all non-commuting diagrams have such a problem.

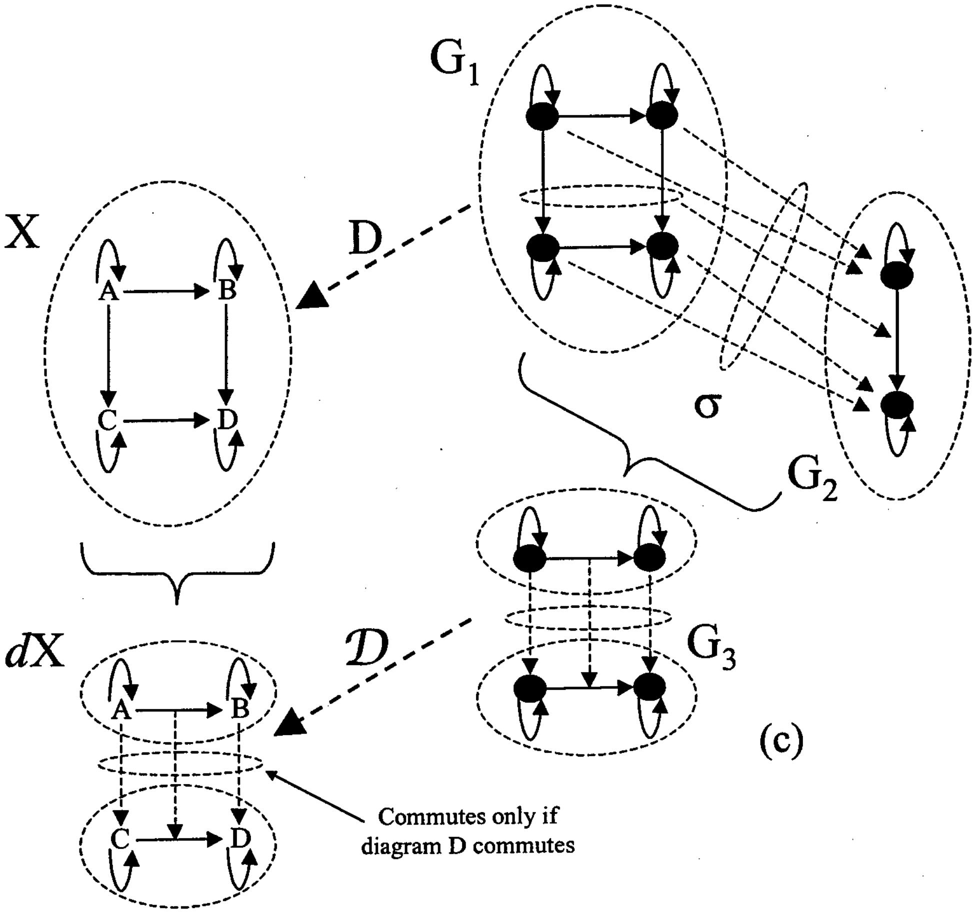
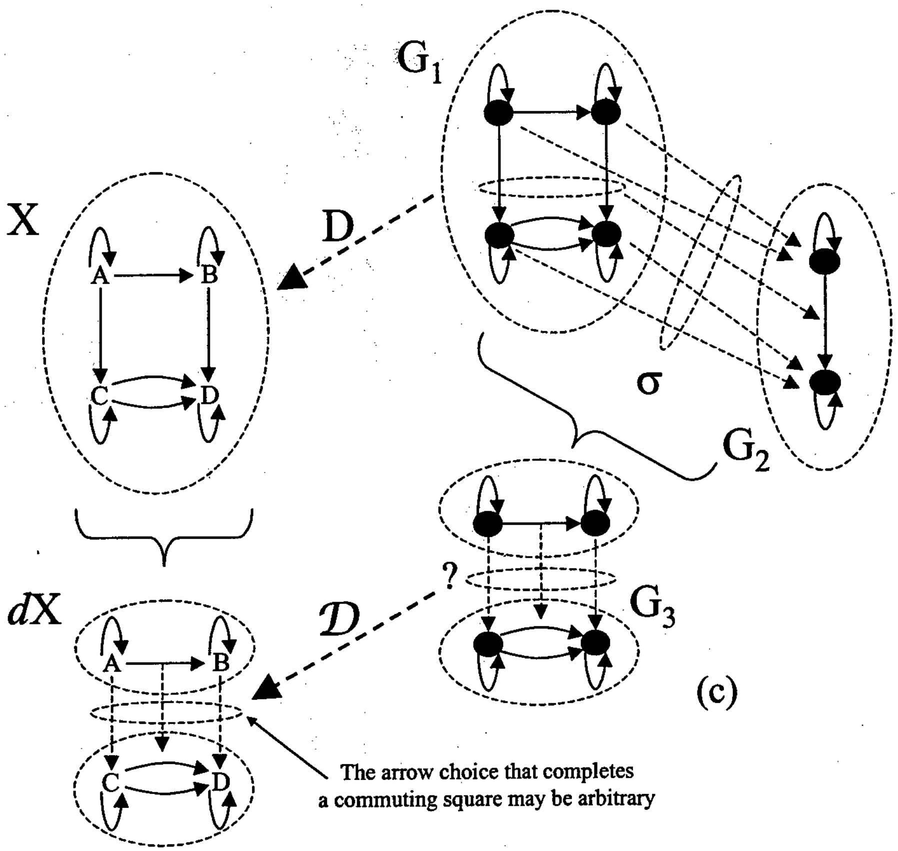


Figure 3-24. The *d*X diagram 𝒟 induced by a partition morphism σ may not exist

The *d*X diagram returned by the partition function may be ambiguous if the non-commuting arrows are within a *d*X object and not between the objects. In Figure 3-25, if the A→B arrow forms a commuting square with both of the C→D arrows then there exist two possible *d*X diagrams for the given partition morphism. The "correct" choice of C→D arrow to pair with the A→B arrow cannot be induced from the partition morphism alone. Either or both choices may be correct but the partition morphism cannot be used to indicate which C→D arrow should be used, hence the ambiguity. Not all non-commuting diagrams have this problem either.



The arrow choice that completes a commuting square may be arbitrary

Figure 3-25. The *d*X diagram $\mathcal{D}$ induced by a partition morphism σ may be ambiguous

Given a category X diagram D:G→X and a partition morphism σ:G→G_0, the category *d*X diagram $\mathcal{D}$:G_0→*d*X can be constructed as follows:

For each $\alpha_i \in$ Objects(G_0), let the *d*X object D_i:G_i→X, be induced by the following assignments:
Objects(G_i) = { $\eta \mid \eta \in$ Objects(G) $\wedge\ \sigma(\eta) = \alpha_i$ },
i.e. all G nodes mapped to node α_i in G_0
Arrows(G_i) = { $f \mid f \in$ Arrows(G) $\wedge\ \sigma(f) = id_{\alpha_i}$ },
i.e. all G arrows mapped to the identity arrow of node α_i in G_0

Note that for each $\eta \in$ Objects(G_i) there exists an associated $\eta \in$ Objects(G) and for each $f \in$ Arrows(G_i) there exists an associated $f \in$ Arrows(G_i).

Let functor D_i be the pair of functions < $D_{i\text{-Objects}}$, $D_{i\text{-Arrows}}$ > defined as follows
For each $\eta \in$ Objects(G_i), $D_{i\text{-Objects}}(\eta)$ = D(η)
For each $f \in$ Arrows(G_i), $D_{i\text{-Arrows}}(f)$ = D(f)

Objects($\mathcal{D}$) = {D_i}, i.e. for each $\alpha_i \in$ Objects(G_0), $\mathcal{D}(\alpha_i)$ = D_i:G_i→X.

For each $f_j \in$ Arrows(G_0),
let the *d*X Arrow $\mathcal{D}(f_j)$ = <δ_j, σ_j>:D_m→D_n,
where D_m:G_m→X = $\mathcal{D}$(dom(f_j))
and D_n:G_n→X = $\mathcal{D}$(cod(f_j)),
be computed as follows:

```
For each node $\eta \in \mathrm{Objects}(G_m)$,
    let $\sigma_j(\eta) = \nu$
        where $\eta \rightarrow \nu \in \mathrm{Arrows}(G) \wedge \sigma(\eta \rightarrow \nu) = f_j$
        and where for all $\alpha \in \mathrm{Objects}(G_n)$,
            where $\eta \rightarrow \alpha \in \mathrm{Arrows}(G)$ and $\sigma(\eta \rightarrow \alpha) = f_j$,
        there exists an arrow $\nu \rightarrow \alpha \in \mathrm{Arrows}(G)$
    Let $\delta_j(\eta) = D(\eta \rightarrow \nu)$
```

Note that the shape arrow $\eta \rightarrow \nu$ must exist because of predicate (1),
the arrow $\eta \rightarrow \nu$ is unique because of predicate (3),
and the shape arrows $\nu \rightarrow \alpha$ must exist because of predicate (4) from Definition 3.19.

```
For each arc $g \in \mathrm{Arrows}(G_m)$,
    let $\delta_j(g)$ be the pair of X arrows $< D_m(g), D_n(g') >$
        where $g'$ is the arc $\sigma_j(\mathrm{dom}(g)) \rightarrow \sigma_j(\mathrm{cod}(g))$
        such that category X arrows $D_m(g)$, $D_n(g')$, $\delta_j(\mathrm{dom}(g))$, $\delta_j(\mathrm{cod}(g))$
            form a commuting square
    Let $\sigma_j(g) = g'$
```

Note that an arc g' is guaranteed to exist for each arc g because of predicate (2).
If a commuting square cannot be found then partition is undefined (see Figure 3-24)
If multiple commuting squares are defined then the *d*X diagram returned by the construction is arbitrary (see Figure 3-25).

Operations Flatten and Partition have a relationship similar to that of functor *Diagramize* and *Colimit*. Flatten takes category *d*X diagrams to category X diagrams and Partition does the reverse. Every category *d*X diagram has only one X diagram to which it will flatten, whereas every category X diagram may have more than one category *d*X diagram to which it will partition.

3.5 Operations over Diagrams

In order to work with category X diagrams on a higher level of abstraction there needs to exist a core set of operations for creating, combining and manipulating category X diagrams instead of just constructing them out of a collection of X-objects and X-arrows. This section defines the concept of a diagram extension as well as operations for joining and folding diagrams. This section also defines the notion of a parameterized diagram and an instantiation of a parameterized diagram. Together these operations are used to define the semantics of the diagram statement syntax presented in Chapter 5, which is a higher-level syntax for creating complex diagrams by building upon smaller component diagrams.

3.5.1 Extending a diagram

Extending a category X diagram involves adding additional nodes and arcs to a diagram while leaving the original nodes and arcs and their associated category X objects and arrows intact.

Definition 3.21. An *extension morphism* in category *d*X is a diagram monomorphism, $<\delta, \sigma>: D_1 \to D_2$, that satisfies the following predicate: $D_2 \circ \sigma = D_1$.

$$\begin{tikzcd}
D_1 = D_2 \circ \sigma & G_1 \arrow[dl, "D_1"'] \arrow[d, "\delta"] \arrow[dr] \arrow[rr, "\sigma"] & & G_2 \arrow[dl, "D_2"] & G_1 \arrow[rr, "\sigma", "Monomorphism"'] \arrow[dr, "D_1"'] & & G_2 \arrow[dl, "D_2"] \\
X & X^{\to} \arrow[l, "Source"] \arrow[r, "Target"] & X & & & X &
\end{tikzcd}$$

Figure 3-26. Definition of a diagram extension

Notation: Diagram D_2 is referred to as an extension of diagram D_1. In an extension morphism $<\delta, \sigma>: D_1 \to D_2$, Objects($\delta$) = $\{id_\alpha \mid \alpha \in \text{Objects(D)}\}$ and Arrows(δ) = $\{<f, f> \mid f \in \text{Arrows(D)}\}$, i.e. the diagram δ objects are the identity arrows for all of the objects in diagram D and the diagram δ arrows are all of the pairs of arrows in diagram D. The two diagrams in Figure 3-26 are equivalent because δ is entirely determined by the source diagram D and is independent of the extension; therefore the diagram on the right commutes.

3.5.2 Joining diagrams

The ultimate form of diagram composition is the colimit operation of category *d*X (Section 3.6) that can combine an arbitrary diagram of diagrams into a new diagram. A less powerful form of composition, actually a special case of the colimit operation, merely unites unrelated diagrams into a single unified diagram. Any number of diagrams can be joined together by taking the disjoint union of their shapes and underlying X-objects and X-arrows.

Definition 3.22. The operation *Join* is a function from a collection of *d*X objects, $\{D_1, D_2, \dots D_n\}$, to a *d*X object $\mathcal{D}$ defined as follows:

$$\begin{aligned}
&\text{Join}(D_1, D_2, \dots D_n) = \text{Flatten}(\mathcal{D}),\\
&\quad \text{where Objects}(\mathcal{D}) = \{D_i \mid 0 \le i \le n\} \text{ and}\\
&\qquad\quad \text{Arrows}(\mathcal{D}) = \{id_{D_i} \mid 0 \le i \le n\}
\end{aligned}$$

The join operation is a special case of the colimit operation of category dX as the collection of objects D_i form a category dX diagram $\mathcal{D}$ with no arrows between the objects. The join operation induces an extension morphism from each component diagram, D_i, to the "joined" diagram, $\mathcal{D}$.

3.5.3 Folding a diagram

The fold operation combines the indicated nodes and arcs of a category X diagram when the underlying associated X-objects and X-arrows of the indicated nodes and arcs are the same X-objects and X-arrows.

Definition 3.23. The operation *fold* is a function from a category X diagram, D_1:G_1→X, and a shape epimorphism σ:G_1→G_2 to a category X diagram D_2:G_2→X, defined by the commuting diagram in Figure 3-27. The fold operation fold(D:G→X, σ) is defined whenever σ is an epimorphism that satisfies the following properties:

For each α, $\beta \in$ Objects(G), where $\sigma(\alpha) = \sigma(\beta)$,
  $D(\alpha) = D(\beta)$
For each f, $g \in$ Arrows(G), where $\sigma(f) = \sigma(g)$,
  $D(f) = D(g)$.

$$\begin{tikzcd}
G_1 \arrow[rr, "\sigma", "\textit{Epimorphism}"'] \arrow[dr, "D_1"'] & & G_2 \arrow[dl, "D_2 = \mathit{fold}\,(D_1, \sigma)"] \\
& X &
\end{tikzcd}$$

Figure 3-27. Definition of fold operation

An example of a fold operation, fold(D_1, < σ combing nodes 3 and 5 >), is depicted in Figure 3-28. The nodes labeled 3 and 5 have the same underlying category X-object, C, and therefore the fold operation is defined over the given inputs. A different σ could combine nodes 1 and 2 with or without also combining the arcs labeled α and β.

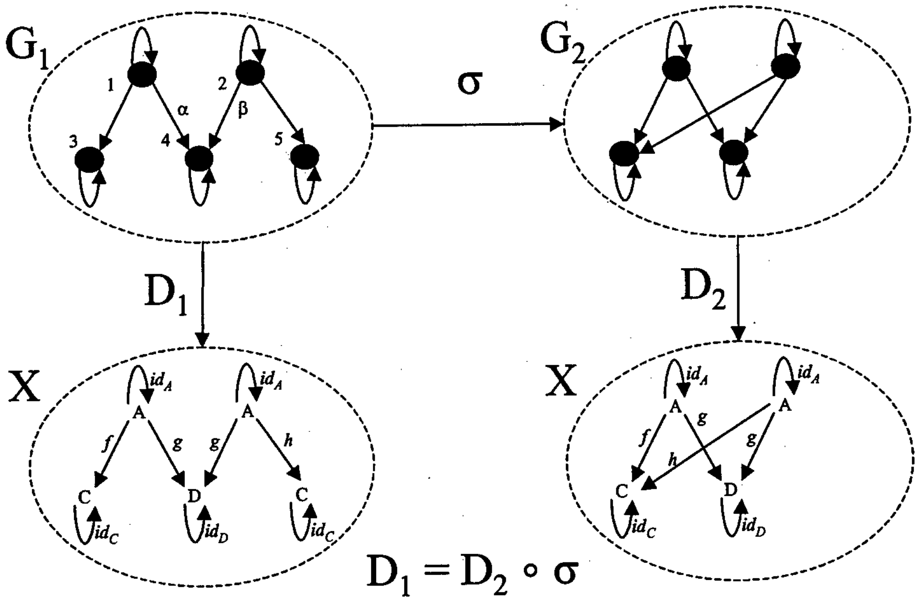


Figure 3-28. Example fold operation

3.5.4 Diagram parameterization

An extension morphism does not limit or direct how a diagram can be extended. The concept of parameterization controls or guides the possible extensions of a diagram when coupled with a suitable form of instantiation.

Definition 3.24. A *parameterized diagram* is a pair <D, σ> where D is a category X diagram D:G→X and where σ is a shape monomorphism σ:G→G'.

$$\begin{tikzcd}
G \arrow[r, "\sigma"] \arrow[d, "D"'] & G' \\
X &
\end{tikzcd}$$

Figure 3-29. Definition of a parameterized diagram

One can view a parameterized diagram as having a fixed part and a variable part. The fixed part is the shape G, the variable part is the part of shape G' that is not covered by shape G. Figure 3-30 depicts a simple parameterized diagram, <D:G→X, σ:G→G'>, where only one node in G' (and the accompanying arrow to that node and identity arrow for that node) is uncovered. Object A is called the formal parameter and object B is called the body.

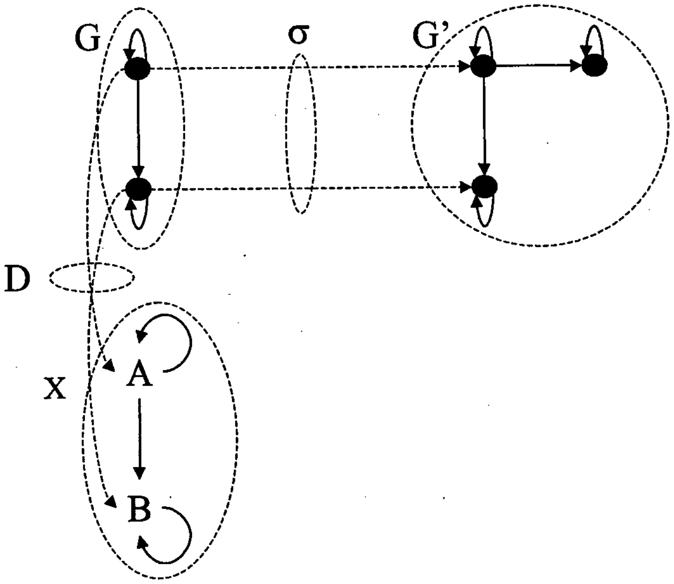


Figure 3-30. Example parameterized diagram

3.5.5 The "shape" of a parameterized diagram

It can be argued that the expressive power of parameterized diagrams comes from the difference between shapes G and G' and not the underlying assignment of shape G to category X objects and arrows. Given a parameterized diagram < D:G→X, σ:G→G' >, shape G represents the underlying fixed objects and arrows from category X, and the parts of shape G' not covered by shape G represent the variable/unknown/parameterized objects and arrows from category X. The depiction in Figure 3-31 captures the "interesting" part of the parameterized diagram from Figure 3-30; namely, the difference between shapes G and G'. The solid circles and lines represent shape G and the fixed or known category X objects and arrows and the dashed circles and lines represent the difference between shape G and G', the parameterized part. Shape G' can always be indicated this way as σ:G→G' is a monomorphism.

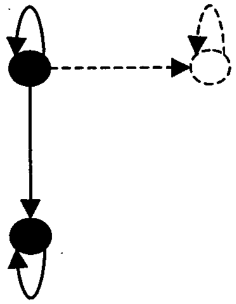


Figure 3-31. Underlying "shape" of the example parameterized diagram

3.5.6 Diagram instantiation

Instantiating a parameterized diagram involves "fixing" the variable part while leaving the fixed part alone.

Definition 3.25. An *instantiation* of a parameterized category X diagram <D:G→X, σ:G→G'> is a category X diagram D':G'→X such that there exists an extension morphism *f* :D→D'.

An instantiation of a parameterized diagram <D:G→X, σ:G→G'> is an extension diagram of D with shape G', see Figure 3-32. One can think of diagram D' as diagram D with additional arcs and nodes (and category X objects and arrows) "added on" to diagram D such that it has the shape G'.

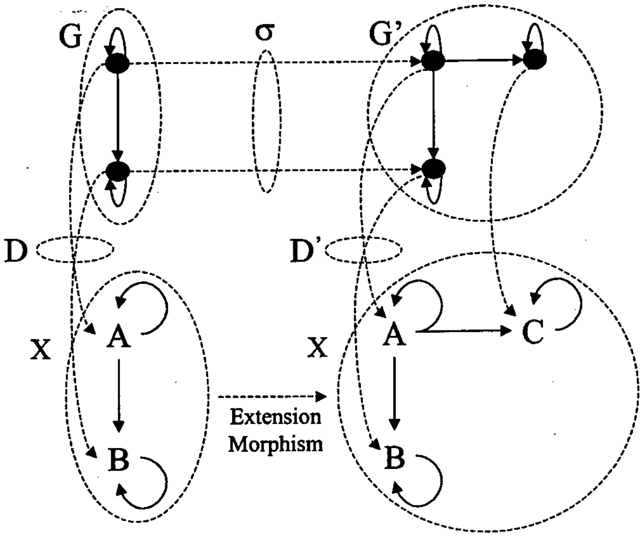


Figure 3-32. Instantiation of a parameterized diagram

3.5.7 Analysis of parameterized diagrams

The notion of a parameterized diagram, < D, σ >, places no restrictions on the relationship between the known and unknown parts. For example, an unknown node could be the source of an unknown arrow to a known target or it could be the target of unknown arrows from a known source. During instantiation the unknown node (and arrow) must be associated with some category X object (and arrow); thus the orientation of the known and unknown parts is critical to determining which X-objects and X-arrows (if any) can be used to instantiate the parameterized diagram. Table 3-1 depicts a selection of differently oriented parameterized diagrams and analyzes them as to the "range" of X-objects and X-arrows that can be used to instantiate the parameterized diagram.

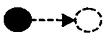
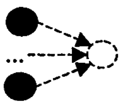
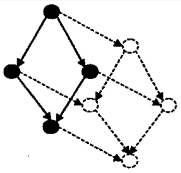
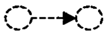
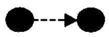
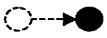
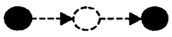
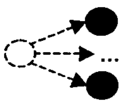
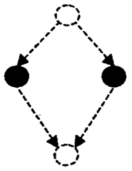
Parameterized Diagram (fragments) and their meaning in terms of strict instantiation		
Parameterized Diagram	**Meaning** **Solid = known/fixed** **dashed = unknown/variable/parameterized**	**Can be (strictly) instantiated by...**
	The unknown object must have at least as much structure as the known object.	The identity arrow and the given object are the lower bound, no upper bound
	The unknown object must have at least as much structure as the objects from which it has arrows	The colimit of the sub-diagram whose sink is the unknown object and the associated cocone morphisms are the lower bound, no upper bound
	The unknown diagram must have at least as much structure as the known diagram.	Identity arrows and the given objects (and the arrows between them) serve as the lower bound, no upper bound, fixing any individual unknown object may influence the other objects
	There are no restrictions on the unknown object	Any object
	There are no restrictions on the unknown objects or the arrow between them	Any arrow and associated objects, fixing either object alone restricts the remaining object
	The unknown arrow must be between the two known objects	There may be no instantiation
	The unknown object can have at most as much structure as the known object.	The empty object and the initial arrow (if any) are the lower bound, the given object and the identity arrow are the upper bound
	The unknown object must have at least as much structure as the known "source" object and no more structure than the known "target" object	If there exists an arrow between the "source" and "target" objects then the source and target objects form the lower and upper bounds for the unknown object, otherwise there is no instantiation
	The unknown object must (collectively) have no more structure than each individual object to which it has an arrow	The initial object and the initial arrow (if they exist) are the lower bound, the limit (if it exists) of the diagram whose source is the unknown object and the associated cone arrows are the upper bound.
	The unknown objects are interdependent in that instantiating either may influence the structure of the other	Fixing either object may influence the other

Table 3-1. Analysis of parameterized diagram shapes

In Table 3-1 the first three rows are lower bound parameterizations where the unknown part is required to have an arrow to it from the known part and not vice versa. The second part of the table, after the break, contains more diverse notions of parameterization, including those where the known parts play the role of an upper bound on the unknown parts instead of as a lower bound. In the syntax to be developed in Chapter 5, only the lower bound form of parameterization is used, i.e. where the "formal parameter" part places a requirement on the "actual parameter" part.

3.6 Colimits in the Category of Diagrams

In this section a colimit operation over category *d*X diagrams is developed. This operation enables category *d*X diagrams (diagrams of diagrams of category X) to be combined in a way that preserves both the content and the structure of the underlying category X diagrams in a minimal fashion. This operation is used in Chapter 6 to compose and apply design information.

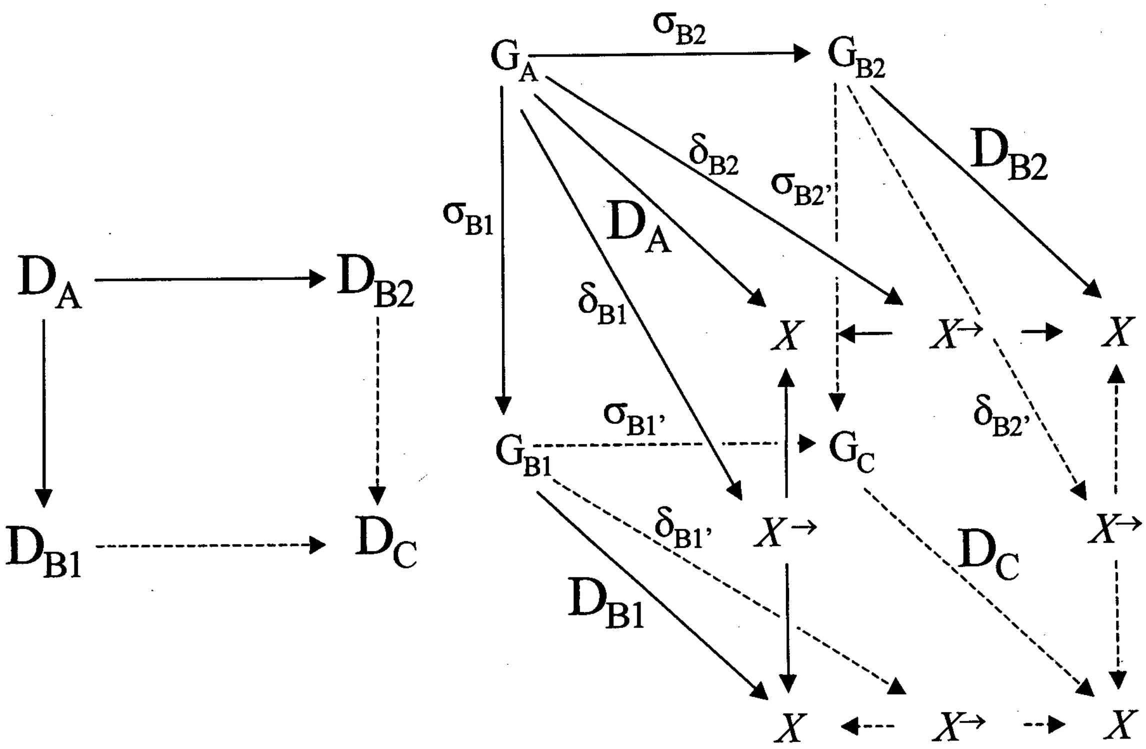
The sub-sections of this section are as follow: Section 1 defines an operation over pushout diagrams of category *d*X. Section 2 contains an example of the pushout operation in use. Section 3 proves that the operation defined in Section 1 is a is a pushout operation for select *d*X diagrams. Section 4 contains a proof that category *d*X has colimits for select diagrams, and Section 5 contains a direction for extending the proof of colimits to encompass all diagrams of category *d*X.

3.6.1 The Pushout operation in category *d*X

A pushout (Definition A.15) in category *d*X, depicted in Figure 3-33(a) actually has the underlying structure depicted in Figure 3-33(b).

Definition 3.26. Given a category *d*X diagram $D_{B1} \leftarrow D_A \rightarrow D_{B2}$ and given that category X is cocomplete, an object D_C and commuting arrows $D_{B1} \rightarrow D_C \leftarrow D_{B2}$ can always be determined using the following steps:

(1) Determine the shape diagram, G_C, along with functors σ_{B1}', and σ_{B2}'.
(2) Create the functor D_C:$G_C \rightarrow$X by labeling nodes and arcs of shape G_C with category X objects and arrows.
(3) Determine functors δ_{B1}' and δ_{B2}' in the pair of arrows D_{B1} —<δ_{B1}', σ_{B1}'>→ D_C and D_C←<δ_{B2}', σ_{B2}'>— D_{B2} so that the diagram in Figure 3-33(b) commutes.



a. Pushout in *d*X

b. Pushout when a *d*X object is viewed as a functor D:G →X and a *d*X arrow is viewed as the pair of functors <δ, σ>

Figure 3-33. Underlying structure of a category *d*X pushout

(1) Given, $D_{B1} \leftarrow\!\!<\delta_{B1}, \sigma_{B1}\!\!>\!\!— D_A —\!\!<\delta_{B2}, \sigma_{B2}\!\!>\!\!\rightarrow D_{B2}$, as in Figure 3-33(b), construct shape G_C, and functors σ_{B1}', and σ_{B2}' as the pushout object and arrows of the shape category diagram $G_{B1} \leftarrow G_A \rightarrow G_{B2}$ (Proposition 3.4). The back square of Figure 3-33(b) is now complete. The domain of functor D_C is known to be shape category G_C. Step 2 uses Shape G_C and diagram $D_{B1} \leftarrow D_A \rightarrow D_{B2}$ to determine the pushout object D_C.

(2) The objects of category X in thẹ image of (functor) diagram D_C are constructed by taking the colimit of a "cover" diagram, that consists of select X-objects and X-arrows from (and between) category X diagrams D_A, D_{B1}, and D_{B2}. The arrows of category X diagram D_C are the universal arrows between a colimit object of a cover diagram and a cocone object of a cover diagram.

The diagram functor D_C:$G_C \rightarrow X$ is the pair of functions < D_{Object}, D_{Arrow} > defined as follows:

Definition of D_{Object}:

For each $\alpha \in$ Objects(G_C),
$D_{Object}(\alpha)$ = Colimit$_X$(D_α), where the cover diagram D_α is defined as follows:

Let C-Nodes(α) = { dom(f) | $f \in$ Arrows(G_C) and cod(f) = α}
i.e. the nodes of G_C that have arrows to α.

Let A-Nodes(α) = {η | $\eta \in$ Objects(G_A), and ($\sigma_{B1}' \circ \sigma_{B1}$)($\eta$) $\in$ C-Nodes(α)}
Let B1-Nodes(α) ={η | $\eta \in$ Objects(G_{B1}), and $\sigma_{B1}'(\eta) \in$ C-Nodes(α)}
Let B2-Nodes(α) ={η | $\eta \in$ Objects(G_{B2}), and $\sigma_{B2}'(\eta) \in$ C-Nodes(α)}
i.e. the nodes in G_A, G_{B1}, G_{B2} that are mapped to the C-nodes.

Let A-Objects(α) = {$D_A(\eta)$ | $\eta \in$ A-Nodes(α)}
Let B1-Objects(α) = {$D_{B1}(\eta)$ | $\eta \in$ B1-Nodes(α)}
Let B2-Objects(α) = {$D_{B2}(\eta)$ | $\eta \in$ B2-Nodes(α)}
i.e. the X objects underlying the indicated G_A, G_{B1}, and G_{B2} nodes.

Let A-Arrows(α) = { $D_A(f)$ | $f \in$ Arrows(G_A) and dom(f), cod (f) $\subseteq$ A-Nodes(α)
Let B1-Arrows(α) = { $D_{B1}(f)$ | $f \in$ Arrows(G_{B1}) and dom(f), cod (f) $\subseteq$ B1-Nodes(α)
Let B2-Arrows(α) = { $D_{B2}(f)$ | $f \in$ Arrows(G_{B2}) and dom(f), cod (f) $\subseteq$ B2-Nodes(α)
i.e. the X arrows underlying the arcs between the indicated G_A, G_{B1}, G_{B2} nodes.

Let δ-B1-Arrows(α) = {$\delta_{B1}(\eta)$ | $\eta \in$ A-Nodes(α)}
Let δ-B2-Arrows(α) = {$\delta_{B2}(\eta)$ | $\eta \in$ A-Nodes(α)}
i.e. the X-arrows underlying the diagram morphism associated with the indicated nodes.

Objects(D_α) = A-Objects(α) $\cup$ B1-Objects(α) $\cup$ B2-Objects(α)
Arrows(D_α) = A-Arrows(α) $\cup$ B1-Arrows(α) $\cup$ B2-Arrows(α) $\cup$
δ-B1-Arrows(α) $\cup$ δ-B2-Arrows(α) $\cup$ identity arrows
and closed under composition

Definition of D_{Arrow}:

For each $f \in$ Arrows(G_C),
$D_{Arrow}(f)$ = the universal arrow between D_{Object}(dom(f)) and D_{Object}(cod(f)),
which are the colimit object and a cocone object respectively of the cover diagram $D_{dom(f)}$.

As a further explanation of D_{Arrow}, there is an extension morphism $D_{dom(f)} \rightarrow D_{cod(f)}$ between the cover diagrams $D_{dom(f)}$ and $D_{cod(f)}$ for each $f \in$ Arrows(G_C). Because of this extension morphism the proof of the existence of a universal arrow depicted in Figure 3-18 applies.

Note that because the X-objects of diagram D_C are constructed using the colimit operation and the X-arrows of D_C are universal arrows, diagram D_C must be a commuting diagram whether or not diagrams D_A, D_{B1}, or D_{B2} are commuting diagrams.

In Figure 3-33(b) the functors σ_{B1}', σ_{B2}', and D_C (depicted as dashed arrows) have been determined. To complete the diagram in Figure 3-33(b), functors δ_{B1}' and δ_{B2}' are constructed in Step 3.

(3) The functor δ_{B1}' is the pair of functions $<\delta_{Object}, \delta_{Arrow}>:G_{B1}\rightarrow X^{\rightarrow}$ defined as follows:

For each $\alpha \in$ Objects(G_{B1}),
$\delta_{Object}(\alpha)$ = the colimit cocone arrow $D_{B1}(\alpha)\rightarrow D_C(\sigma_{B1}'(\alpha))$.

This arrow occurs in the colimit diagram of the cover diagram $D_\beta G_\beta\rightarrow X$, where $\beta = \sigma_{B1}'(\alpha)$, because $D_C(\sigma_{B1}'(\alpha))$ = colimit(D_β), and $\alpha \in$ Objects(G_β), and therefore $D_{B1}(\alpha) = D_\beta(\alpha)$.

For each $f \in$ Arrows(G_{B1}),
$\delta_{Arrow}(f) = <s, t>$
where $s \in$ Arrows($D_{B1}(G_{B1})$) such that dom(s) = D_{B1}(dom(f)) and cod(s) = D_{B1}(cod(f)),
$t \in$ Arrows($\sigma_{B1}' \circ D_C$)(G_C) such that dom(t) = ($\sigma_{B1}' \circ D_C$)(dom(f)) and
cod(t) = ($\sigma_{B1}' \circ D_C$)(cod(f)).

Note that the functor $\sigma_{B1}':G_{B1}\rightarrow G_C$ and Step (2) guarantees that a "t" X-arrow exists for each "s" X-arrow. Because the "t" X-arrow is a universal arrow and a cocone arrow of a diagram involving the "s" X-arrow the X-arrow pair $<s, t>$ along with the X-arrows δ_{Object}(cod(f)) and δ_{Object}(dom(f)) form a commuting square. Note also that the term ($\sigma_{B1}' \circ D_C$) is the functor from shape G_{B1} to the restriction of diagram D_C, thus ($\sigma_{B1}' \circ D_C$)(dom(f)) and ($\sigma_{B1}' \circ D_C$)(cod(f)) are X-objects in diagram D_C.

Functor $\delta_{B2}':G_{B2}\rightarrow X^{\rightarrow}$ is similarly constructed over functors $D_{B2}:G_{B2}\rightarrow X$, $\sigma_{B2}':G_{B2}\rightarrow G_C$, and $D_C:G_C\rightarrow X$. The diagram in Figure 3-33(b) is now complete.

The diagram $D_{B1} \leftarrow<\delta_{B1}, \sigma_{B1}>— D_A —<\delta_{B2}, \sigma_{B2}>\rightarrow D_{B2}$ with constructed object $D_C:G_C\rightarrow X$ and arrows $D_{B1}—<\delta_{B1}', \sigma_{B1}'>\rightarrow D_C \leftarrow<\delta_{B2}',\sigma_{B2}'>—D_{B2}$ form a commuting square by construction.

3.6.2 Example pushout in category *d*X

An example category *d*X diagram, $D_{B1}\leftarrow D_A\rightarrow D_{B2}$ in Figure 3-34, aids in visualizing how the operation defined in Definition 3.26 constructs the *d*X-object D_C and *d*X-arrows $< \delta_{B1}',\sigma_{B1}' >:D_{B1}\rightarrow D_C$, $< \delta_{B2}',\sigma_{B2}' >:D_{B1}\rightarrow D_C$. The diagram in Figure 3-34 is a specific example of the generic structure depicted in Figure 3-33. The labels in diagrams D_A, D_{B1}, and D_{B2} in Figure 3-34 are the actual designations of the X-objects in the diagram. The labels in diagram D_C are merely placeholder names as the actual X-objects are to be determined by the construction.

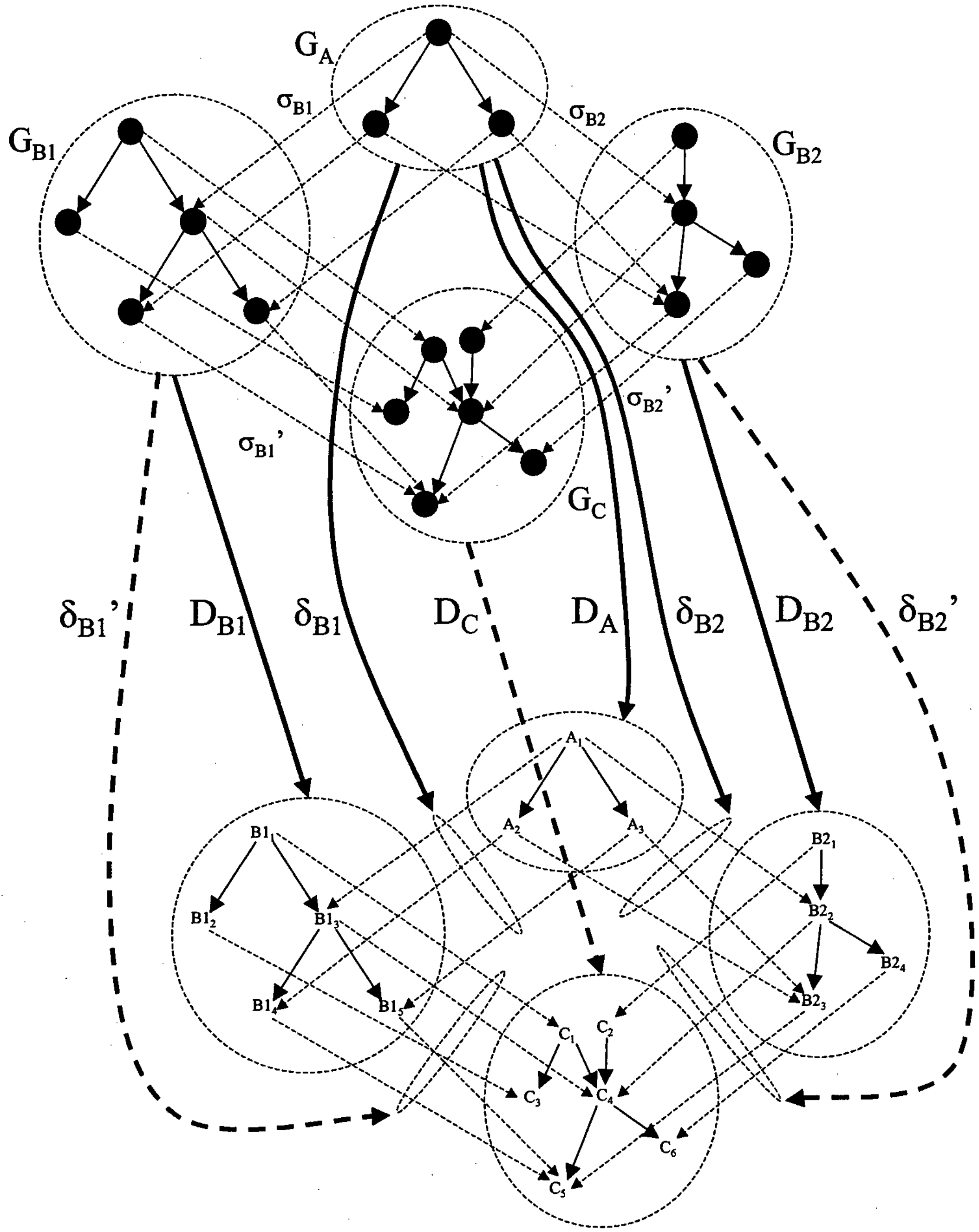


Figure 3-34. Example category *d*X pushout

Step (1): In Figure 3-34, shape G_C and shape morphisms σ_{B1}', and σ_{B2}' are determined by taking the pushout of the category of shapes diagram $G_{B1} \leftarrow G_A \rightarrow G_{B2}$. Thus, the domains of the diagram functors $D_C:G_C \rightarrow X$, $\delta_{B1}':G_{B1} \rightarrow X^{\rightarrow}$, and $\delta_{B2}':G_{B2} \rightarrow X^{\rightarrow}$ are known after this step but the actual X-arrows and X-objects in the codomain of diagram functor D_C and the $X^{\rightarrow}$-arrows and $X^{\rightarrow}$-objects in the codomain of diagram functors δ_{B1}' and δ_{B2}' are unknown (hence the thick dashed arrows representing those functors).

Step (2): The cover diagrams that are used to construct the X-objects and X-arrows of the diagram D_C in Figure 3-34 are defined as follows:

Objects(D_{C_1}) = {$B1_1$} and Arrows(D_{C_1}) = { id_{B1_1} }.
Objects(D_{C_2}) = {$B2_1$} and Arrows(D_{C_2}) = { id_{B2_1} }.
Objects(D_{C_3}) = {$B1_1$, $B1_2$} and Arrows(D_{C_3}) = {$B1_1 \rightarrow B1_2$, id_{B1_1}, id_{B1_2} }.
Objects(D_{C_4}) = {A_1, $B1_1$, $B1_3$, $B2_1$, $B2_2$} and
Arrows(D_{C_4}) = {$A_1 \rightarrow B1_3$, $A_1 \rightarrow B2_1$, $B1_1 \rightarrow B1_3$, $B2_1 \rightarrow B2_2$, *and identity arrows*}.
Objects(D_{C_5}) = Objects(D_{C_4}) ∪ {A_2, A_3, $B1_4$, $B1_5$, $B2_3$} and
Arrows(D_{C_5}) = Arrows(D_{C_4}) ∪ { $B1_3 \rightarrow B1_4$, $B1_3 \rightarrow B1_5$, $B2_2 \rightarrow B2_3$, *and id arrows*}.
Objects(D_{C_6}) = Objects(D_{C_4}) ∪ {$B2_4$} and
Arrows(D_{C_6}) = Arrows(D_{C_4}) ∪ {$B2_2 \rightarrow B2_4$, id_{B2_4} }.

The X-objects of diagram D_C are determined by taking the colimit of the cover diagram of that X-object. The (trivial) colimit of diagram D_{C_1} is object $B1_1$. Thus node C_1 in diagram D_C Figure 3-34 is the X-object $B1_1$. The node labeled C_2 in diagram D_C is colimit(D_{C_2}), which is object $B2_1$. The node labeled C_3 is colimit(D_{C_3}), which is object $B1_2$. The node labeled C_4 is colimit(D_{C_4}), which is equivalent to the pushout object of diagram $B1_3 \leftarrow A_1 \rightarrow B2_2$. Finally, the nodes labeled C_5 and C_6 are the colimit objects of their respective cover diagrams, D_{C_5} and D_{C_6} .

The X-arrows of diagram D_C are the universal arrows between a colimit object and any other cocone object. Note that if there is an arrow $\alpha \rightarrow \beta$ in D_C (as indicated by the shape G_C) then the cover diagram D_β is an extension of the cover diagram D_α. (The X-objects and X-arrows of the cover diagram of node C_4 are a subset of those of the cover diagram of node C_5.) Thus there is a universal arrow $\alpha \rightarrow \beta$ induced by the colimit and cocone arrows of the cover diagram D_α to α and β respectively.

Step (3): The category $X^{\rightarrow}$ diagrams δ_{B1}':$G_{B1} \rightarrow X^{\rightarrow}$ and δ_{B2}':$G_{B2} \rightarrow X^{\rightarrow}$ are determined in the following manner: The objects in G_{B1} and G_{B2} are mapped to the cocone X-arrows formed by the colimit of the category X cover diagrams. The arrows in G_{B1} and G_{B2} are mapped to the related pairs of X-arrow in diagrams (D_{B1} and D_C) and (D_{B2} and D_C) that must necessarily form commuting squares with the X-arrows previously assigned to the objects of G_{B1} and G_{B2}.

As an example, the X-arrow $B1_3 \rightarrow C1_4$ is the cocone arrow to node C_4 formed by the colimit of the cover diagram of node C_4 (of which $B1_3$ is an object). The X-arrow $B1_4 \rightarrow C_5$ is formed similarly. Thus

X-arrows $B1_3 \rightarrow C_4$ and $B1_4 \rightarrow C_5$ are objects of the category $X^{\rightarrow}$ diagram δ_{B1}'. The pair of X-arrows $B1_3 \rightarrow B1_4$ and $C_4 \rightarrow C_5$ form a commuting diagram with X-arrows $B1_3 \rightarrow C_4$ and $B1_4 \rightarrow C_5$ and are thus assigned as an arrow of the category $X^{\rightarrow}$ diagram δ_{B1}'. Every object and arrow of the category $X^{\rightarrow}$ diagram δ_{B1}' (and δ_{B2}') can be assigned in such a manner.

3.6.3 Category *d*X has pushouts for all Nice diagrams

While the operation defined in Section 3.6.1 is defined for all category *d*X diagrams of the form $D_{B1} \leftarrow D_A \rightarrow D_{B2}$, it is not the pushout operation for all such *d*X diagrams. Specifically, the operation defined in Section 3.6.1 is the pushout operation for "Nice" *d*X diagrams that when flattened (Definition 3.18) yields a commuting category X diagram.

Definition 3.27. A category *d*X diagram $\mathcal{D}$ is termed *Nice* if Flatten($\mathcal{D}$) is a commutative diagram.

Proposition 3.28. Category *d*X has pushouts for all Nice diagrams of the form $D_{B1} \leftarrow D_A \rightarrow D_{B2}$ (assuming category X is cocomplete).

Proof: The operation defined Section 3.6.1 in Definition 3.26 constructed a category *d*X object D_C and the arrows to it from a given category *d*X diagram $D_{B1} \leftarrow D_A \rightarrow D_{B2}$ in such a manner that a commuting square was formed. The constructed object must be the minimal such object to form a commuting square in order for the operation to be designated as the pushout operation (of Nice diagrams) of category *d*X. Specifically, any other *d*X object, D_X, that also forms a commuting square with the diagram $D_{B1} \xleftarrow{<\delta_{B1},\, \sigma_{B1}>} D_A \xrightarrow{<\delta_{B2},\, \sigma_{B2}>} D_{B2}$, must have a unique arrow to it from the constructed object D_C, namely the arrow $<\delta_U, \sigma_U>$ depicted in Figure 3-35.

$$\begin{tikzcd}
D_A \arrow[r] \arrow[d] & D_{B2} \arrow[d, dashed] \arrow[ddr] & \\
D_{B1} \arrow[r, dashed] \arrow[drr] & D_C \arrow[dr, dotted, "< \delta_U, \sigma_U >"] & \\
 & & D_X
\end{tikzcd}$$

Figure 3-35. Required universal arrow for object D_C to be the pushout object

Minimality of the constructed pushout object D_C: Assume that a *d*X object D_X:$G_X \rightarrow X$ exists that forms a commuting square with the given pushout diagram. A unique arrow $<\delta_U, \sigma_U>$:$D_C \rightarrow D_X$ is constructed as follows:

The shape morphism σ_U of the arrow $<\delta_U, \sigma_U>$ is the universal arrow between the colimit object G_C and a cocone object G_X of the shape category diagram $G_{B1} \leftarrow G_A \rightarrow G_{B2}$. (Because shape G_C is the pushout of the shape category diagram $G_{B1} \leftarrow G_A \rightarrow G_{B2}$, and shape G_X must have a cocone of shape morphisms to it from that same diagram, there must be a unique shape category morphism from G_C to G_X.)

The shape category morphism, σ_U:$G_C \rightarrow G_X$, is used to associate every X-object and X-arrow within diagram D_C with an associated X-object and X-arrow within diagram D_X. Each X-object in diagram D_C is, by construction, the colimit of a category X (cover) diagram made up of X-objects and X-arrows in (and between) diagrams D_A, D_{B1}, and D_{B2}. The associated X-object in diagram D_X must be that same colimit object or it must be the target of some other cocone morphism from that same cover diagram in order for D_X to be part of a commuting diagram. In either case there is a unique (universal) X-arrow from each (colimit) X-object in diagram D_C to the associated (cocone) X-object in diagram D_X.

For each $\alpha \in$ Objects(G_C),
$\delta_{U\text{-Object}}(\alpha) = f$, where f:$D_C(\alpha) \rightarrow D_X(\sigma_U(\alpha))$ is the universal arrow between a colimit object, $D_C(\alpha)$, of the cover diagram D_α and a cocone object, $D_X(\sigma_U(\alpha))$, of that same diagram.

The X-objects and X-arrows in diagrams D_C all commute with the X-objects and X-arrows inside (and between) diagrams D_A, D_{B1}, and D_{B2} by construction. The X-objects and X-arrows in diagram $D_X \circ \sigma_U$ all commute with diagrams D_A, D_{B1}, and D_{B2} because D_X is assumed to be part of a commuting diagram.

For each $f \in$ Arrows(G_C),
$\delta_{U\text{-Arrow}}(f) = < D_C(f), (D_X \circ \sigma_U)(f)>$.

Thus $\delta_U(f), f \in$ Arrows(G_C) is the pair of X-arrows, one from diagram D_C and one from diagram D_X, that [along with the universal X-arrows $\delta_{U\text{-Object}}(\text{dom}(f))$ and $\delta_{U\text{-Object}}(\text{cod}(f))$] complete a commuting square in category X.

If diagram D_X has additional structure beyond the minimum needed based on the shape category G_C then the same logic applies as the extra "structure" of diagram D_X is ignored by the restriction $D_X \circ \sigma_U$. If diagram D_X has "collapsed" some of the requisite structure based on shape G_C (shape morphism

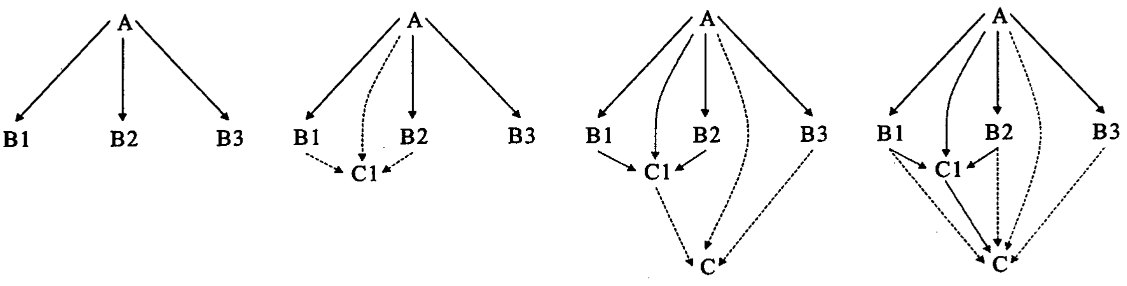
σ_U:$G_C \rightarrow G_X$) then the "collapsed" X- objects within diagram D_X will still have cocone arrows to them from all of the cover diagrams for which the D_C specification has colimit arrows and the same logic applies. The $D_C \rightarrow D_X$ arrow is guaranteed to be unique as the individual X-arrows making up the $D_C \rightarrow D_X$ arrows connecting the diagrams are unique. □

3.6.4 Category *d*X has colimits for all Nice diagrams

Any category that has an initial object and all pushouts is cocomplete [Gol84]. Because the operation defined in Section 3.6.1 is the pushout operation only for Nice *d*X diagrams it cannot be used to prove that category *d*X is cocomplete. However, the operation can be used to prove that all category *d*X diagrams that are Nice have colimits.

Proposition 3.29. The category *d*X has colimits for all Nice diagrams if category X is cocomplete.

Proof: The empty *d*X diagram is the initial object for category *d*X and it is a Nice diagram in category *d*X. Because the colimit of a diagram of a category (with an initial object) can always be formed by taking successive pushouts (as depicted in Figure 3-36), all Nice category *d*X diagrams have colimits by successively applying the operation defined in Section 3.6.1. □



a. Initial diagram b. 1st pushout c. 2nd pushout d. Colimit

Figure 3-36. Constructing a colimit using successive pushouts

The operation defined in Section 3.6.1 was specifically tailored to operate over category X diagrams of the form $D_{B1} \leftarrow D_A \rightarrow D_{B2}$. It can easily be extended (made more generic) to operate over all category *d*X diagrams by developing cover diagrams that encompass the entire *d*X diagram.

3.6.5 Colimits and commuting vs. non-commuting category *d*X diagrams

This section describes some of the issues with developing a colimit operation over non-Nice diagrams of category *d*X. This section represents a direction for future work.

Given a Nice diagram $\mathcal{D}:G_0 \rightarrow dX$ in category *d*X, its colimit object $D:G \rightarrow X$ is guaranteed to be a commuting diagram. This means that if there exists a pair of arcs between two nodes in shape G that these arcs will be assigned the same underlying category X arrow by functor (diagram) D. This assignment of the same X-arrow occurs because the colimit of a covering diagram is used to construct the X-objects associated with the nodes and a universal arrow is used to construct the X-arrow(s) associated with the arc(s) between the nodes. For Non-Nice diagrams the colimit object D may not be a commuting diagram and therefore the colimit operation and the universal arrow cannot be used to construct the underlying X-objects and X-arrows of the colimit object.

As a trivial example, the colimit of the singleton non-commuting diagram on the left in Figure 3-37, where $f \neq g$, should be itself. Unfortunately, the commuting diagram on the right in Figure 3-37, where Object C is a "collapsed" object B, is constructed using the colimit operation as defined in this section. The reason is that the cover diagram of "node" C is a non-commuting diagram ($A \rightrightarrows B$) and therefore taking the colimit of the cover diagram will eliminate structure that under certain situations should be retained. One way to solve this problem is to use an operation other than the colimit operation on the covering diagram.

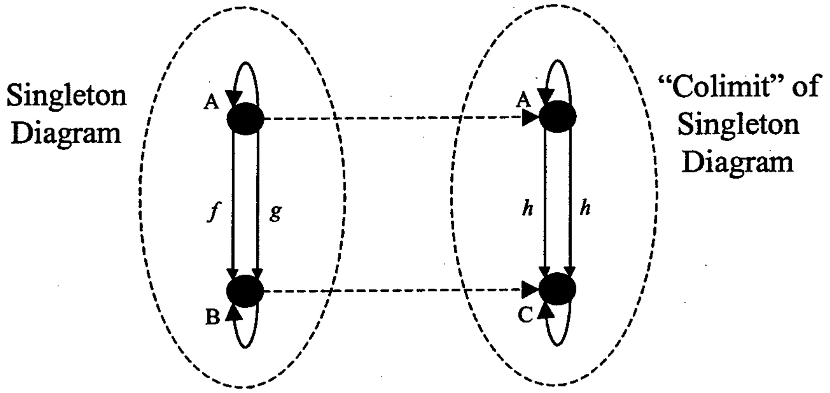


Figure 3-37. Colimits of non-commutative diagrams

As an example of that operation, in the category X diagram in Figure 3-38 the object C is the colimit object, and the object D is not the target of a cocone as $f' \neq g'$ and yet $(f' \circ \alpha) = (\beta \circ f)$,

$(g' \circ \alpha) = (\beta \circ g)$, and there is a unique arrow D→C. Thus object D has retained some of the structure that has been lost in object C in that diagram.

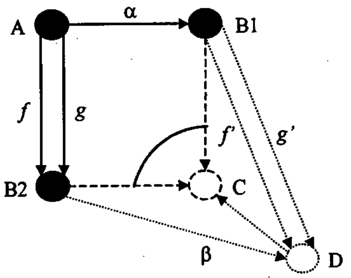


Figure 3-38. Pushout-like structure-preserving operation

Object D can be constructed as depicted in Figure 3-39. On the left in Figure 3-39, two pushouts, using arrow *f* and then arrow g, construct objects D_f and D_g. On the right the pushout of diagram $D_f \leftarrow B2 \rightarrow D_g$ constructs object D. Also on the right, composing arrows through Df and Dg formed non-commuting arrows f 'and g'. The arrow to the colimit object C is guaranteed to exist as the colimit of the diagram on the right in Figure 3-39 (including the object D) is object C. It may be that this operation, when defined over larger non-commuting diagrams, will serve as the appropriate operation to apply to the covering diagram that enables the proof of cocompleteness to be extended to all category *d*X diagrams.

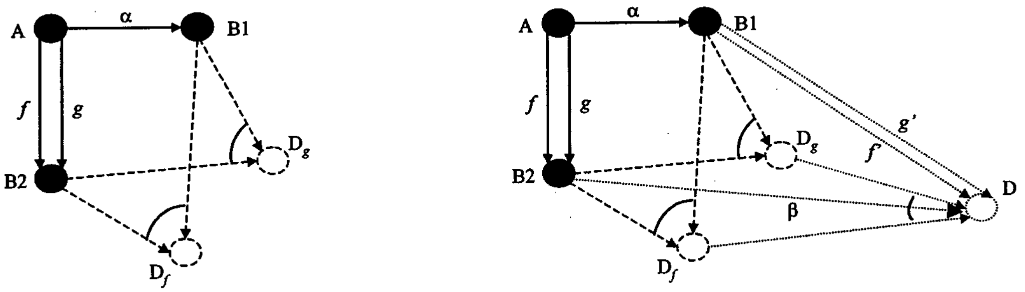


Figure 3-39. Construction of "structured pushout"

An alternative approach to proving that category *d*X is cocomplete may be to use a different covering diagram than the maximal one used by the construction defined in this section. For example, in the trivial example in Figure 3-38 each node could be used as its own covering diagram. This "minimal" covering diagram may be difficult to determine for more complex non-commuting diagrams.

3.7 Contributions and Future Work

The main contributions of this chapter are the definition of a morphism between diagrams of an arbitrary category, and the definition of a diagram category *d*X based on diagrams of an arbitrary category. The category *d*X enables diagrams and diagram morphisms to be treated as objects and arrows within a category and is a generalization of the work done in [SJ95]. A diagram morphism is useful as it ensures that both the structure and the content of one diagram is preserved in another.

The proof that the category of shapes is cocomplete is original and fairly straightforward. The proof of the colimit operation over Nice diagrams in category *d*X is original as well and of greater significance. Even though the full category *d*X has not been proved to be cocomplete, colimits over the Nice *d*X diagrams are sufficient for their use in this dissertation in later chapters.

The secondary contributions of this chapter are the operations and functors defined over the categories X, *d*X and the diagrams of those categories. The functors are *Diagramize* and *Colimit*. The operations are *Flatten, Partition*, *Join*, and *Fold*. Also of note is the definition of an extension morphism in the category *d*X along with the notions of parameterization and instantiation.

The notion of parameterized diagrams developed in this chapter is very weak but can be extended when applied to diagrams of a particular category such as category *Spec* in Chapter 4. The operations together with the notion of parameterization and instantiation are used in Chapter 5 to formally define the semantics of a syntax for constructing and manipulating diagrams on a higher level.

Future work involves extending the pushout and colimit operations (and proofs) for diagram categories to extend over all the diagrams of category *d*X instead of just those over Nice diagrams.

4 The Category of Specification Diagrams

This chapter applies the diagram theory developed in Chapter 3 to the category *Spec* presented in Appendix A in order to derive the category *dSpec*. This category has diagrams of specifications and morphisms as the objects and diagram morphisms as the arrows. All of the operations and functors developed for category dX are applicable to category dSpec. This chapter also defines several properties and operations of the objects, arrows and diagrams of category *dSpec* that are used in later chapters. Thus this chapter develops the theory underlying category *dSpec* that is not generic to all diagram categories.

Many of the properties defined over the category *dSpec* arrows in this chapter are based on similar properties defined over category *Spec* arrows. This chapter also applies the theory underlying parameterized diagrams, Section 3.5.4, to diagrams in category *Spec* and then further extends that theory based on the properties of category *Spec* and *dSpec*. Finally, this chapter develops the notion of a diagram interpretation that forms the basis for representing design information in Chapter 6.

Section 4.1 defines category *dSpec*, defines what it means for a *dSpec* arrow to be a conservative or definitional extension morphism and defines the model semantics for these objects and arrows. Section 4.2 defines what it means to be a parameterized diagram in *dSpec*. Section 4.3 defines diagram interpretations, morphisms between diagram interpretations and the category *dInterp*. This section also defines the model semantics of diagram interpretations and compares them to interpretations and diagram refinement.

4.1 The Category of Spec Diagrams, dSpec

The category *dSpec* is the diagram category (Proposition 3.13) derived from category *Spec*. Its objects are diagrams of category *Spec* (Proposition B.19) and its arrows are diagram morphisms (Definition 3.12) between the *Spec* diagrams.

The following functors and operations are defined over category *dSpec* and category *Spec* and the diagrams of those categories

Diagramize: *Spec*→*dSpec* (Proposition 3.15).

Colimit: *dSpec*→*Spec* (Proposition 3.17).

Flatten: *dSpec*-diagram → *Spec*-diagram (Definition 3.18).

Partition: *Spec*-diagram × Partition-morphism → *dSpec*-diagram (Definition 3.20).

Nice diagrams (Definition 3.27) of category *dSpec* have colimits (Proposition 3.29).

As category *dSpec* objects and arrows are related to category *Spec* objects and arrows via the *Colimit*: *dSpec*→*Spec* functor, the model semantics (Section B.2) of category *dSpec* objects and arrows can be defined in terms of the model semantics of the associated category *Spec* objects and arrows. Thus *dSpec* objects have the same model semantics as their associated *Spec* objects as defined in Section B.2.5 and *dSpec* arrows have the same model semantics as their associated *Spec* arrows as defined in Section B.2.6. Each *dSpec* object has an associated class of models and each *dSpec* arrow has an associated reduct functor that takes target models to source models by forgetting the unneeded carrier sets and functions.

4.1.1 Conservative and definitional extension arrows in *dSpec*

Given a category *dSpec* diagram $\mathcal{D}$:G→*dSpec*, the functor *Colimit*: *dSpec*→*Spec* yields a category *Spec* diagram with the same shape G, i.e. *Colimit* ∘ $\mathcal{D}$:G→*dSpec* = D:G→*Spec*. As every *dSpec*-arrow has an associated *Spec*-arrow via the *Colimit* functor, *dSpec* arrows can be classified as conservative extensions or definitional extensions based on the classification of the colimit of the arrow in category *Spec*.

Definition 4.1. A morphism, $D_1 \rightarrow D_2$, in *dSpec* is a *conservative extension* morphism or a *definitional extension* morphism in *dSpec* if the *Spec* arrow *Colimit*($D_1 \rightarrow D_2$) is a conservative extension morphism or a definitional extension morphism in *Spec* respectively.

Notation: The labeled arrows –c→ and –d→ in *dSpec* diagrams indicate the property of being a conservative or definitional extension morphism, the same as they do for *Spec* (Definition C.8 and Definition C.12).

Proposition 4.2. Conservative and definitional extension morphisms are closed under composition in *dSpec*.

Proof: If A–x→B and B–x→C are conservative extensions or definitional extensions in *dSpec* then *Colimit* (A–x→B) and *Colimit* (B–x→C) are conservative extensions or definitional extensions in *Spec* by definition. Since those properties are closed under the composition operation in *Spec* (Proposition C.15) and *Colimit* (A–x→B) ∘ *Colimit* (B–x→C) is equal to *Colimit* (A–x→B ∘ B–x→C) these properties are closed under composition in *dSpec*. □

Pushouts in category *Spec* were proven to have preserved conservative and definitional extension morphisms (Proposition C.19). The same is true for category *dSpec* morphisms.

Proposition 4.3. Pushouts in *dSpec* preserve conservative and definitional extension morphisms. (Those properties reflect across a pushout square.)

Proof: Functor *Colimit* takes a *dSpec* pushout diagram to a *Spec* pushout diagram. Functor *Colimit* takes the *dSpec* pushout object to the associated *Spec* pushout object. Therefore, since conservative extensions and definitional extensions are preserved in pushouts in *Spec*, they must also be preserved in pushouts in *dSpec* □

Category *Spec* morphisms that are conservative extensions and definitional extensions were defined so as to establish reduct functors that had properties that were useful for describing the refinement of specifications and describing parameterization. These same properties hold for category *dSpec* objects and arrows and the refinement and parameterization of diagrams of specifications. Thus for conservative extension morphisms in *dSpec*, each model associated with the source diagram can be extended, sometimes in many different ways, so as to be a model of the target diagram. (As described in Section C.1.4 for *Spec* conservative extension morphisms.) Also, for definitional extension morphisms in *dSpec*, each model associated with the source diagram can be extended in a unique way so as to be a model of the target diagram. (As described in Section C.1.5 for *Spec* definitional extension morphisms.)

4.1.2 *dSpec* diagrams

Category *dSpec* diagrams are diagrams of category X diagrams; these multiple levels can be confusing so this section introduces a notation to keep the confusion to a minimum.

An example diagram in *dSpec* that contains a definitional extension morphism is depicted in Figure 4-1. The figure depicts two levels of diagrams: *Spec* diagrams and *dSpec* diagrams. Identity arrows were left off of the two diagrams to make them less complicated. On the right in Figure 4-1 two simple *Spec* diagrams Bag (One-Sort→Bag) and SetAsBag (One-Sort → SetAsBag) are joined together by a diagram morphism that connects the "parameters" (specification One-Sort) and the "bodies" (specification

Bag and Specification SetAsBag) of the diagram. On the left in that figure the two *Spec* diagrams (Bag and SetAsBag) are viewed as *dSpec* objects and the diagram morphism between them becomes a *dSpec* arrow. The *dSpec* arrow is a definitional extension morphism as *Colimit*(Bag→SetAsBag) is a definitional extension morphism in *Spec*.

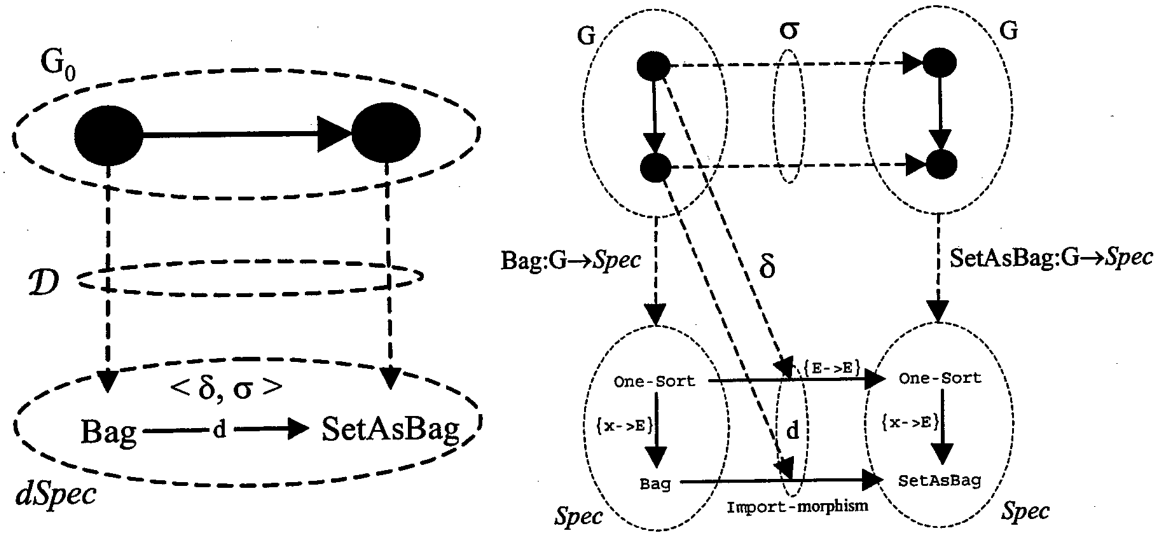


Figure 4-1. Example *dSpec* diagram and underlying *Spec* diagrams and diagram morphisms

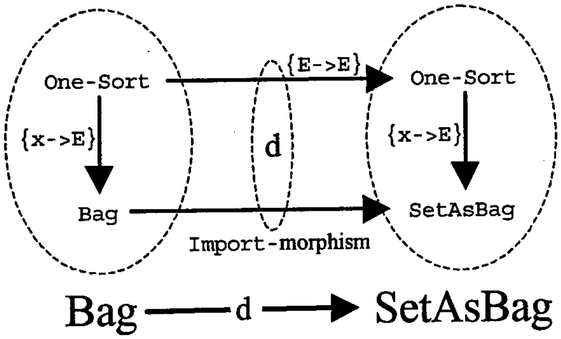


Figure 4-2. Single diagram representing all of the detail depicted in Figure 4-1

In the remainder of the document, in figures of *d*X diagrams, the relationship between the shape category and the target category and the relationship between *dSpec* diagrams and *Spec* diagrams will be subsumed using a single picture. For example, the *dSpec* diagram and underlying *Spec* diagrams in Figure 4-1 will be depicted as in Figure 4-2 where the dashed ovals represent *dSpec* objects and arrows and the contents of the ovals represent the underlying *Spec* diagrams and diagram morphisms. (Note that in Figure 4-1 and Figure 4-2 the terms "Bag" and "SetAsBag" are being used to name both a diagram and a specification within that diagram.) The shapes of the *Spec* and *dSpec* diagrams can be inferred from what is depicted in Figure 4-2 as long as only Nice diagrams are depicted.

4.1.3 Constructing conservative and definitional extension morphisms in *dSpec*

Just as building upon previously existing specifications can extend objects in *Spec* (Section C.1.2), building upon previously existing objects in *dSpec* can extend objects in *dSpec*. A simple way to extend a *dSpec* object, D_1, so that its extension, D_2, has one of the named arrow properties in Definition 4.1 is depicted in Figure 4-3. In Figure 4-3 the *Spec* object C, which is $\text{colimit}_{Spec}(D_1)$, is extended conservatively or definitionally to create a *Spec* object F. The resultant *Spec* object is then treated as a *dSpec* object, i.e. $D_2 = \textit{Diagramize}(F)$. The *dSpec* arrow $D_1 \rightarrow D_2$ is simply the colimit cocone arrows individually composed with the extension arrow, C→F. As the *Spec* arrow C→F was constructed to be one of the indicated extension morphisms, the *dSpec* arrow $D_1 \rightarrow D_2$ is also classified as such an arrow.

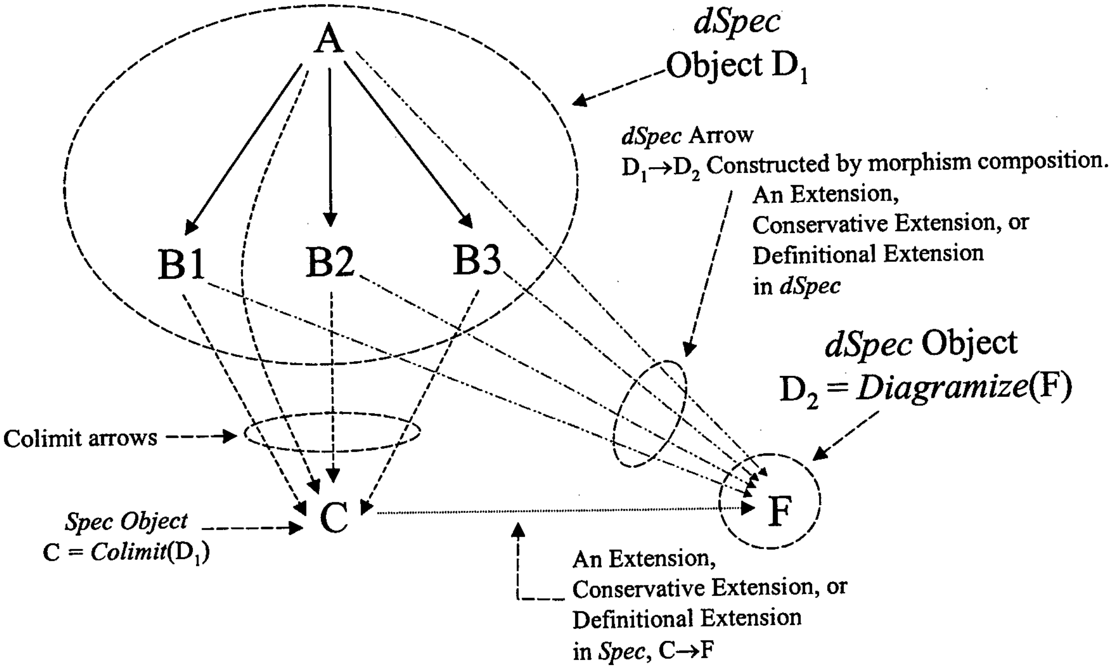


Figure 4-3. Creating (conservative, definitional) extensions in *dSpec*

The construction depicted in Figure 4-3 is guaranteed to result in a diagram morphism between diagrams as the colimit induced cocone morphisms to specification C commute, and therefore the related cocone morphisms to specification F commute as well.

4.1.4 Specifying diagram morphisms

In order to develop a diagram morphism, $< \delta, \sigma >:D_1 \rightarrow D_2$, one must establish which *Spec* arrows connect the specification objects of the source diagram to the specification objects of the target diagram. Under certain conditions associating the specification morphisms between the "nodes" of the diagrams is enough to induce a diagram morphism between the diagrams. The syntax presented in Figure 4-4 establishes a morphism between diagrams by establishing individual morphisms between the objects (nodes) in the two diagrams.

```
Diagram-Morphism →     dMorphism [ dMorphism-Name : ]
                                   diag-ref -> diag-ref is
     ( import-morphism | Arc ( , Arc )* )

arc →       [ arc-name : ] g-node-ref -> g-node-ref : sig-mapping-term

sig-mapping-term →     { [ sig-map-rule (, sig-map-rule )* ] }
sig-mapping-term →     morphism-ref

sig-map-rule →   ( op-ref -> op-ref ) | ( sort-ref -> sort-ref )
```

Figure 4-4. Syntax for a diagram morphism

Assume that the source *diag-ref* is the diagram $D_1:G_1 \rightarrow Spec$, and the target *diag-ref* is $D_2:G_2 \rightarrow Spec$. If diagram D_2 was constructed by importing D_1 (the diagram statement Section 5.2.2 Figure 5-3), then the term "`import-morphism`" indicates the associated diagram morphism. Otherwise, each individual specification object in D_1 must have a single specification morphism (`arc`) to a specification object in D_2 as indicated by a collection of `Arc` clauses in the diagram morphism statement. The collection of specification morphisms between the specification objects must induce a shape morphism, $\sigma:G_1 \rightarrow G_2$, or the purported diagram morphism statement has no meaning (i.e. an error). If D_1 and D_2 are Nice diagrams (Definition 3.27), then the shape morphism $\sigma:G_1 \rightarrow G_2$ can always be induced. The collection of specification morphisms must also induce a δ diagram, $\delta:G_1 \rightarrow Spec^{\rightarrow}$, whose objects are the collection of specification morphisms and whose arrows are pairs of specification morphisms (one from D_1 and one from D_2) as indicated by the restriction $D_2 \circ \sigma$ or the purported diagram morphism statement has no meaning (i.e. an error). Both of these conditions are easy to check syntactically. Thus, to be a correct diagram morphism statement, the given arcs/specification morphisms in the `dMorphism` statement must induce a diagram morphism, $<\delta, \sigma>:D_1 \rightarrow D_2$.

Figure 4-5 is an example diagram morphism between the Bag diagram, One-Sort → Bag, and the Set as Bag diagram, One-Sort→SetAsBag.

```
diagram Set-as-Bag is
  nodes Set-as-Bag, One-Sort
  arcs
    One-Sort -> Set-as-Bag: {X->E}
end-diagram

dMorphism Bag-to-SetAsBag:
               Bag -> Set-as-Bag is
  One-sort -> One-Sort: {},
  Bag -> Set-as-Bag: {}
```

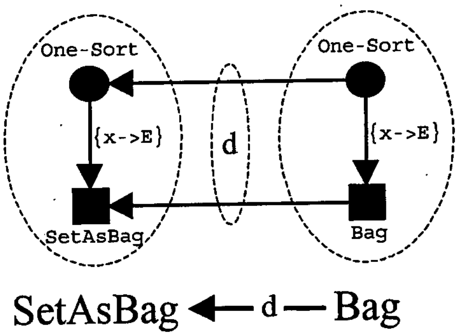


Figure 4-5. Bag to SetAsBag diagram morphism

Figure 4-6 is an example diagram morphism between the Set diagram, One-Sort→Set, and the Set as Bag diagram, One-Sort→SetAsBag.

```
dMorphism Set-to-SetAsBag:
               Set -> Set-as-Bag is
  One-sort -> One-Sort: {},
  Set -> Set-as-Bag: {}
```

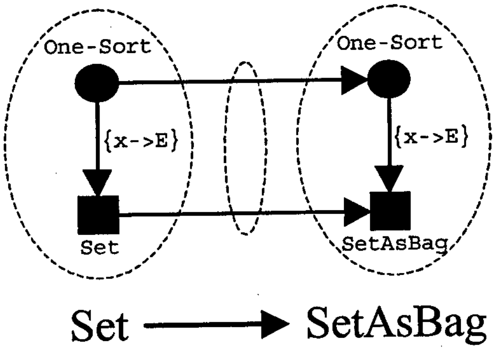


Figure 4-6. Set to SetAsBag diagram morphism

4.2 Parameterized Diagrams in dSpec

A parameterized object is an object with a *variable part* and a *fixed part*. When dealing with parameterized diagrams, a *body* is a sub-diagram of a *dSpec* diagram that has a fixed part and a variable part and a *formal parameter* is a diagram that represents and covers the variable part of a given body. There must exist a diagram morphism from the formal parameter part to the body indicating how the formal parameter covers the variable part of the body. These informal descriptions of variable part, formal parameter and body and their relationships with each other are defined in this section.

4.2.1 Definition of variable part, formal parameter and body

Definition 4.4. Any *Spec* diagram D, where the specification Colimit_{Spec}(D) has more than one (non-isomorphic) model, is said to have a *variable part.*

Definition 4.5. Any *Spec* diagram D, where the specification Colimit_{Spec}(D) has only one model (class of isomorphic models), is said to have *no variable part.*

Definition 4.4 and Definition 4.5 use the number of non-isomorphic models associated with a specification in order to determine if a *Spec* diagram has "variable" parts or not. In general, it is difficult to mechanically establish the number of models associated with a specification unless certain conditions are met. For example if all axioms of a specification are equational or conditional equational and initial semantics is used then one can mechanically prove whether or not there exists a single model associated with the specification. The notion of variable parts is used so that one can reason whether a formal parameter covers *all* of the variable parts of a body. This in turn is used so that one can reason whether a refinement of a body has (unknowingly) constrained the formal parameters, or has (unknowingly) introduced additional variable parts.

Definition 4.6. Given a category *dSpec* diagram D_1, a category *dSpec* diagram D_2, and a morphism $D_1 \rightarrow D_2$, the diagram D_1 is said to be a *formal parameter* of the *body* diagram D_2 and to *fully cover* the variable part of D_2 if the following three conditions are met:

1. $D_1 \rightarrow D_2$ is a conservative extension morphism.
2. For each $\alpha \in \text{Objects}(D_1)$, the colimit induced cocone arrow $f: \alpha \rightarrow \text{Colimit}(D_1)$ is a conservative extension morphism.
3. Given that D_X is a diagram with no variable parts, $\text{Colimit}_{Spec}(\text{Flatten}(D_2 \leftarrow D_1 \rightarrow D_X))$ also has no variable parts.

Conservative extension morphisms (Figure 4-7 depicts conditions 1 and 2) ensure that the formal parameter part actually represents some part of the body specification in its entirety, so to speak. Condition 1 ensures that diagram D_2 does not restrict the models associated with diagram D_1 when diagram D_1 is taken as a whole. Condition 2 ensures that diagram D_2 does not restrict the models associated with the individual specifications making up diagram D_1. This ensures that any actual parameter that has a morphism to it from the formal parameter will be compatible with the body and not have its properties be fundamentally changed in some manner in the instantiating diagram. Like the notion of variable parts, the notion of a conservative extension is also difficult to prove mechanically (as alluded to in Section C.1.4). However, there exists a collection of "conservative extension schemes" [GM93] that make the development of a conservative extension (and the related proof of existence) easier.

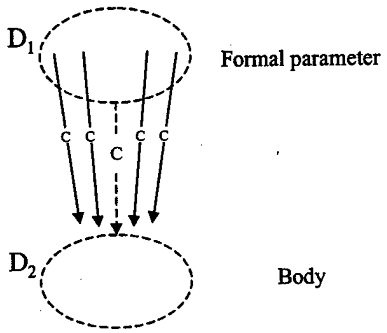


Figure 4-7. Parameterization requires conservative extension morphisms (*Spec* and *dSpec*)

The notion of a variable part existing in the body but not existing in an instantiating diagram (condition 3) ensures that the variable part of diagram D_2 is fully covered by diagram D_1. This ensures that instantiation that involves an actual parameter that has no variable parts will result in a diagram with no variable parts.

4.2.2 Examples of variable parts, formal parameters and bodies

In the next several paragraphs, the terms One-Sort, Flag, Set, Map (in Figure 4-8), Total-Order and Tree (in Appendix F) are all treated as specifications as well as singleton diagrams, and the arrows between them are all treated as morphisms and diagram morphisms. This abuse of terminology helps to avoid having to explicitly use the *Diagramize* (and *Colimit*) functor over those *Spec* (and *dSpec*) objects and arrows when switching between the different meanings of the terms.

The specification One-Sort has many models associated with it, as there are no axioms that limit the associated models. In contrast, the specification Flag has only one model (or class of isomorphic models) associated with it. In every model of specification Flag the carrier set associated with the sort Flag contains exactly three elements (hence they are all isomorphic to each other) and therefore the diagram Flag has no variable parts. The specification Set has many models associated with it as the element sort E has not been "defined", i.e. the sort E could be empty, or could have any number of "values" associated with it that effects how many elements the carrier set of sort Set has associated with it. Therefore the diagram Set has a variable part. The specification Tree has a sort E and an order TO that is not defined and therefore the diagram Tree has a variable part.

```
Spec Empty
-- Spec Empty contains the sorts Nat and Boolean
-- including operations such as:
-- < and > (less than and greater than)
--         of rank Nat, Nat -> Boolean
end-spec
```

```
Spec One-Sort is
  Sort X
end-spec
```

```
Spec Flag is
sort Flag
  op Green:  -> Flag
  op Yellow: -> Flag
  op Red:    -> Flag
constructors {Green, Yellow, Red} construct Flag
  ax Green ≠ Yellow
  ax Yellow ≠ Red
  ax Red ≠ Green
end-spec
```

```
spec Set is
  sort Set, E
  op Empty:          -> Set
  op Insert: E, Set -> Set
  op In:     E, Set -> Boolean
Constructors {Empty, Insert} construct Set
  Forall s:Set, e,e1,e2:E
  Ax not(In(e,Empty))
  Ax in(e1,Insert(e2,s)) ⇒ ((e1=e2) ∨ In(e1,s))
  Ax Insert(e,s) = Insert(e,Insert(e,s))
  Ax Insert(e1,Insert(e2,s)) = Insert(e2,Insert(e1,s))
end-spec
```

```
spec Map is
  sort Map, Dom, Cod
  op Constant-Map:     Cod -> Map
  op Modify: Map, Dom, Cod -> Map
  op Apply:  Map, Dom      -> Cod
Constructors {Constant-Map, Modify} construct Map
  Forall m,m1,m2:Map, d,d1,d2:Dom, c, c1, c2:Cod
  Ax Apply(Constant-Map(c),d) = c
  Ax Apply(Modify(m,d1,c),d2) = if d1=d2 then c else Apply(m,d2)
  Ax Modify(Modify(m,d,c1),d,c2) = Modify(m,d,c2)
  Ax not(d1=d2) ⇒ Modify(Modify(m,d1,c1),d2,c2) =
                  Modify(Modify(m,d2,c2),d1,c1)
  Ax m1=m2 ⇔ Apply(m1,d) = Apply(m2,d)
end-spec
```

Figure 4-8. Code for Empty, One-Sort, Flag, Set and Map specifications

The specification morphism Empty →Set is a conservative extension morphism, however specification Empty does not cover the variable parts of specification Set and therefore diagram Empty is not the formal parameter of diagram Set. In contrast, the diagram One-Sort is a formal parameter of the body diagram Set as the morphism {X→E}:One-Sort → Set is a conservative extension morphism and the diagram Set←One-Sort→D_X will have no variable parts when diagram D_X has no variable parts. Diagram One-Sort is not the formal parameter of diagram Tree as diagram Tree←One-Sort→D_X still has variable parts, i.e. the TO operator in specification Tree could have many different models associated with it as it has not been defined. Diagram Total-Order is the formal parameter of diagram Tree as the diagram Tree ←Total-Order→D_X has no variable parts.

In the preceding examples of parameterized diagrams the body and formal parameter diagrams were single specifications. The example parameterized diagrams in Figure 4-9 are more complex in that they take advantage of having either the formal parameter diagram or the body diagram (or both) contain more than singleton specifications.

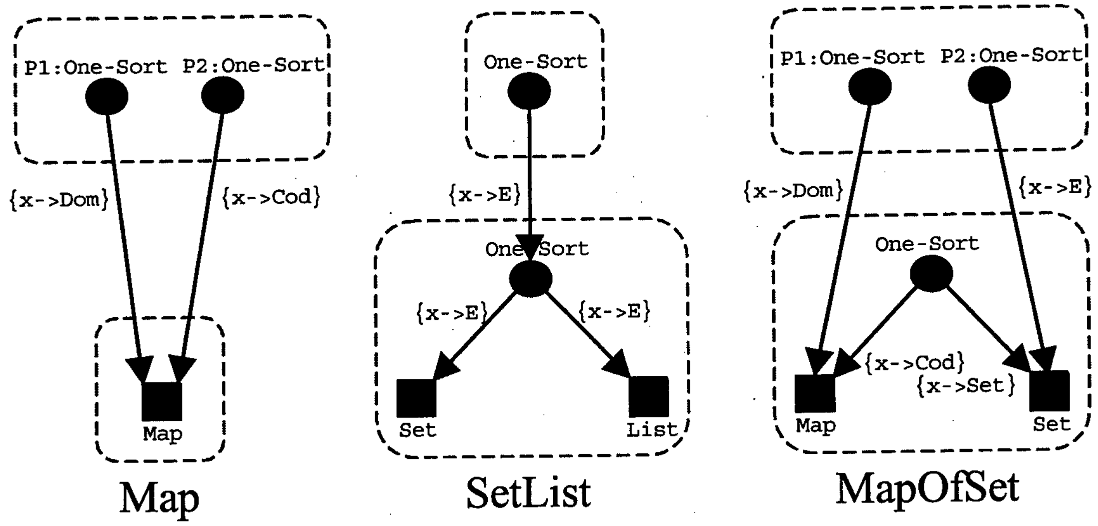


Figure 4-9. Example parameterized diagrams

Notation: Circles in Figure 4-9 represent specifications that traditionally represent the formal parameters of parameterized specifications. Squares represent specifications that traditionally represent the bodies of parameterized specifications.

4.2.3 Parameterized *dSpec* diagrams vs. parameterized diagram theory in *d*X

The generic parameterized diagram theory was introduced in Section 3.5.4,. The theory as presented in that section in Table 3-1 is a good deal richer than is needed for parameterized *Spec* diagrams presented in this chapter. Only lower bound parameterization is used here. Because of this, Figure 3-31 (repeated here in slightly different notation in Figure 4-10) depicts the underlying shape of every category *dSpec* parameterized diagram. The morphism between the formal parameter and body diagrams must be a conservative extension. An instantiation morphism is between the formal parameter diagram and some other diagram that represents the actual parameters. Given the shape of the formal parameter diagram, the shape of the actual parameter diagram can be induced directly. (All specifications and morphisms in the formal parameter diagram must be present in the instantiating diagram. Therefore, ignoring any possible "collapsing" of shape for now, the actual parameter diagram must have the same shape as the formal parameter diagram.)

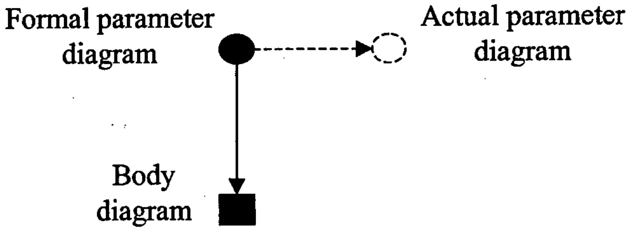


Figure 4-10. Underlying shape of a parameterized diagram in category *dSpec*

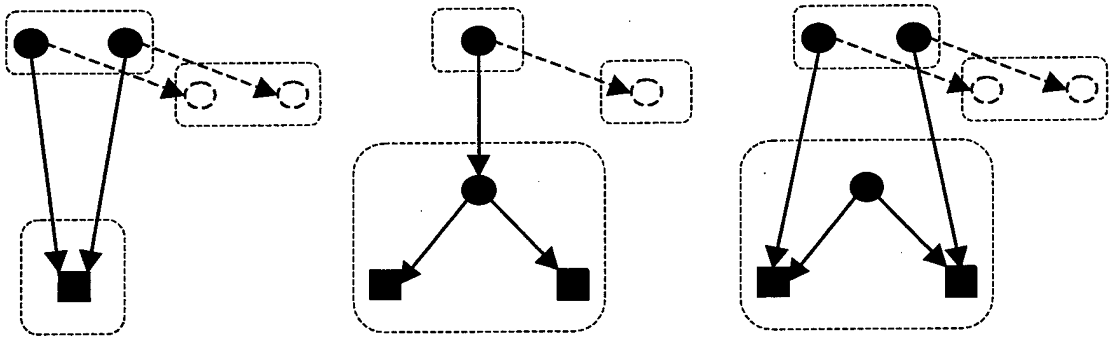


Figure 4-11. Parameterizations in *Spec* have the shape of Figure 4-10 in *dSpec*

Given an actual parameter diagram with the same shape as the formal parameter diagram, the shapes of the parameterized diagrams of Figure 4-9 can be depicted as in Figure 4-11. The dashed boxes could have been left off as the *dSpec* diagram shape (Figure 4-10) can be directly inferred. The notation depicted in Figure 4-11 (without the additional dashed boxes) is used throughout Chapter 5 (and in Appendix E) as the formal parameter and body parts can be determined from the dashed circles and arrows.

4.3 *Diagram Interpretations*

A diagram interpretation is a special diagram in *dSpec* that consists of two diagram morphisms with a common codomain, where one of the diagram morphisms is a definitional extension. Diagram interpretations are similar to the specification interpretations presented in Section C.3 except a diagram interpretation is defined over the category *dSpec* instead of the category *Spec*. In category *dSpec*, the pair of diagram morphisms $S \rightarrow M \leftarrow d - T$ indicates an interpretation of diagrams from source diagram S to target diagram T via the mediator diagram M.

4.3.1 dInterpretations, dInterpretation morphisms and the category *dInterp*

Definition 4.7. An interpretation in *dSpec*, $i = <f, g>$ consists of a diagram morphism f and a diagram definitional extension morphism g with a common codomain diagram, $\mathrm{Cod}(f) = \mathrm{Cod}(g)$. A *dSpec* interpretation, $i = A \rightarrow B^+ \leftarrow d - B$, has operations dom, med, and cod defined as follows: $\mathrm{dom}(i) = A$, $\mathrm{med}(i) = B^+$, and $\mathrm{cod}(i) = B$. These three diagrams are also known as the source, mediator and target diagrams of the *dSpec* interpretation.

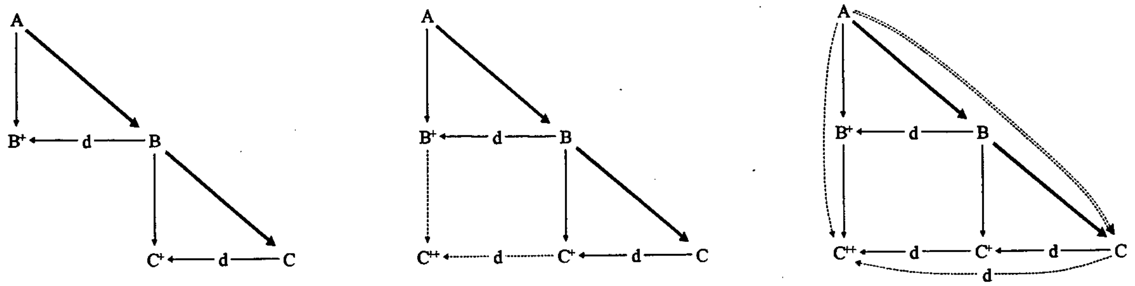
We can visualize the components of a *dSpec* interpretation (also called a *diagram interpretation* or a *dInterpretation*) in Figure 4-12 where the dInterpretation $S \rightarrow M \leftarrow d - T$ has been expanded to $S — <\delta_S, \sigma_S> \rightarrow M \leftarrow <\delta_T, \sigma_T> — T$ where $f = <\delta_S, \sigma_S>$ and $g = <\delta_T, \sigma_T>$.

$$\begin{tikzcd}
G_S \arrow[rr, "\sigma_S"] \arrow[dd] \arrow[ddr, "\delta_S"] & & G_M \arrow[dd] & & G_T \arrow[ll, "\sigma_T"'] \arrow[dd] \arrow[ddl, "\delta_T"'] \\
S \arrow[rr, Rightarrow] & & M & d \arrow[l, Rightarrow] & T \arrow[l, dash] \\
\mathit{Spec} & \mathit{Spec}^{\rightarrow} \arrow[l, "\text{Source}"'] \arrow[r, "\text{Target}"] & \mathit{Spec} & \mathit{Spec}^{\rightarrow} \arrow[l, "\text{Target}"'] \arrow[r, "\text{Source}"] & \mathit{Spec}
\end{tikzcd}$$

Figure 4-12. Structure of an interpretation in category *dSpec*

A diagram interpretation means that the source diagram can be implemented by or refined into a definitional extension of the target diagram. Thus in order to refine the source diagram further one can simply refine the target diagram.

Definition 4.8. *dSpec* interpretations with common codomain/domains compose by taking a pushout and composing the diagram morphisms as depicted in Figure 4-13. This is similar to how interpretations in *Spec* compose in Definition C.23.



a. $A \Rightarrow B \Rightarrow C$ **Two Interpretations**

b. $B^+ \leftarrow B \rightarrow C^+$ **Pushout Construction of C^{++} Mediator**

c. $A \Rightarrow C$ **Interpretation by Morphism Composition**

Figure 4-13. Composition of interpretations in *dSpec*

The two diagram morphisms constructed in Figure 4-5 and Figure 4-6, when joined, constitute a dInterpretation, Set→SetAsBag←d– Bag, as depicted in Figure 4-14. The code for the example interpretation is depicted in Figure 4-15.

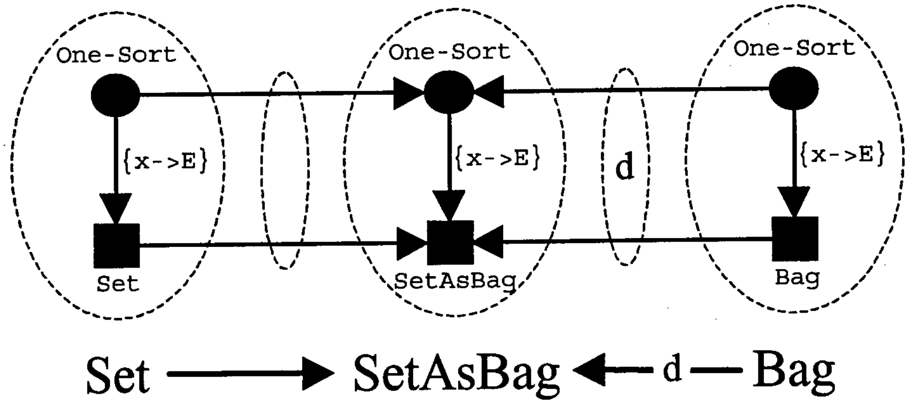


Figure 4-14. Set as Bag diagram interpretation

```
dInterpretation Set-as-Bag: Set => Bag is
  mediator Set-as-Bag
  dom-to-med One-Sort -> One-Sort:{},
             Set -> Set-as-Bag: {}
  cod-to-med One-Sort -> One-Sort:{},
             Bag -> Set-as-Bag: import-morphism
```

Or given the code in Figure 4-5 and Figure 4-6

```
dInterpretation Set-as-Bag: Set => Bag is
  mediator Set-as-Bag
  dom-to-med Set-to-SetAsBag
  cod-to-med Bag-to-SetAsBag
```

Figure 4-15. Set as Bag diagram interpretation code

Definition 4.9. An interpretation morphism in *dSpec* is a 5 tuple, *im* = <$A \Rightarrow B$, $C \Rightarrow D$, f:$A \rightarrow C$, g:$B^+ \rightarrow D^+$, h:$B \rightarrow D$>, where dom(*im*) = $A \Rightarrow B$, cod(*im*) = $C \Rightarrow D$, dom-morphism(*im*) = *f*, cod-morphism(*im*) = *h*, med-morphism(*im*) = *g*, consisting of the domain and codomain interpretations and three morphisms between the domain, codomain and mediator diagrams of the domain interpretation to the domain, codomain and mediator diagrams of the codomain interpretations, such that the diagram in Figure 4-16 commutes.

$$\begin{tikzcd}
\text{Domain Interpretation} \arrow[d, shift right=4] \arrow[d, "\text{Interpretation Morphism}" description] \arrow[d, shift left=4] & A \arrow[r] \arrow[d] & B^+ \arrow[d] & B \arrow[l, "d" description] \arrow[d] \\
\text{Codomain Interpretation} & C \arrow[r] & D^+ & D \arrow[l, "d" description]
\end{tikzcd}$$

Figure 4-16. Interpretation morphism in *dSpec*

Proposition 4.10. Diagram interpretations and diagram interpretation morphisms form a category, *dInterp*.

Proof: The identity arrow of a diagram interpretation is the identity arrows of the Source, Mediator and Target diagrams of the diagram interpretation, i.e. the identity arrow for the interpretation $A \rightarrow B^+ \xleftarrow{d} B$ is the 5 tuple < $A \rightarrow B^+ \xleftarrow{d} B$, $A \rightarrow B^+ \xleftarrow{d} B$, f:$A \xrightarrow{id} A$, g:$B^+ \xrightarrow{id} B^+$, h:$B \xrightarrow{id} B$>. The diagram morphisms (*f*, *g*, *h*) making up a diagram interpretation morphism individually compose and individually are associative, therefore diagram interpretation morphisms compose and are associative. □

Notation: Note the notation has varied on the use of the symbol "*d*" to preface a category. While category *dSpec* is the category of *Spec* diagrams and diagram morphisms, category *dInterp* as defined in Proposition 4.10 is not the category of *Interp* diagrams and diagram morphisms. Instead, category *dInterp* is related to category *dSpec* in the same fashion that category *Interp* is related to category *Spec* (Definition C.20). Since *Interp* is a category, there is a category "*dInterp*" based on diagrams of interpretations, but that category is quite different from the *dInterp* category defined in Proposition 4.10.

Proposition 4.11. Category *dInterp* has functors *Dom*, *Cod*, and *Mod* that return *dSpec* objects and arrows. These functors return the domain, codomain and mediator objects and arrows of *dInterp* objects and arrows as defined in definitions Definition 4.7 and Definition 4.9.

Proof: Essentially the same as the proof of Proposition C.26 for *Dom* , *Cod* and *Mod* functors over category *Interp*. □

4.3.2 Category *dInterp* compared with category *Interp*

Because *dInterp* is not the diagram category of category *Interp* the functors *Diagramize* and *Colimit* defined Section 3.4 Proposition 3.15 and Proposition 3.17 cannot be directly applied to categories *Interp* and *dInterp*. These functors do exist between the two categories but they are based on the functors with the same names between categories *Spec* and *dSpec* and the relationship between those categories and categories *Interp* and *dInterp*.

Proposition 4.12. Category *Interp* has a functor *Diagramize* that takes category *Interp* objects and arrows to category *dInterp* objects and arrows by using the category *Spec* functor *Diagramize* on the individual *Spec* objects and arrows making up an *Interp* object and arrow, i.e. *Diagramize*: *Interp*→*dInterp*.

Proof: An object in *Interp*, $A \Rightarrow B$, is a pair of *Spec* morphisms $A \rightarrow B^{+} \leftarrow B$ with a common codomain where the $B^{+} \leftarrow B$ arrow is a definitional extension. An Arrow in *Interp* is a 5-tuple $im = \langle A \Rightarrow B, C \Rightarrow D, f{:}A \rightarrow C, g{:}B^{+} \rightarrow D^{+}, h{:}B \rightarrow D\rangle$. If the *Spec* functor *Diagramize* is used on the individual *Spec* objects and arrows making up the *Interp* objects and arrows, the result is *dInterp* objects and arrows. The interpretation $A \Rightarrow B$ in *Interp* becomes an interpretation in *dInterp* between (singleton) diagrams A and B with (singleton) diagram B^{+} as the mediator. Similarly, the interpretation morphism *im* in *Interp* becomes an interpretation morphism in *dInterp*.

It is obvious that the identity arrows in *Interp* remain identity arrows in *dInterp* as they do so for the functor *Diagramize:Spec→dSpec*. The *Diagramize:Interp→dInterp* functor preserves the composition of arrows as the *Diagramize:Spec→dSpec* functor does so. □

A *dInterp* object can obviously be viewed as a generalization of an *Interp* object as each *Spec* interpretation is a *dSpec* interpretation via the *Diagramize* functor. However since *Diagramize* is a monic functor and Colimit is not, there are many *dInterp* objects for each *Interp* object and therefore *dInterp* objects can represent more information (via the greater structure inherent in each *dInterp* object). Chapter 6 use this additional structure of diagram interpretations to represent structured design information.

Proposition 4.13. Category *dInterp* has a functor *Colimit* that takes category *dInterp* objects and arrows to category *dInterp* objects and arrows by using the category *dSpec* functor *Colimit* on the individual *dSpec* objects and arrows making up a *dInterp* object and arrow, i.e. *Colimit*: *dInterp→Interp*.

Proof: If each of the *dSpec* objects and arrows making up a *dInterp* object and arrow has the *Colimit*: *dSpec→Spec* functor applied to it the result is an *Interp* object and arrow. It is obvious that identity arrows and arrow compositions are preserved. □

Every *dSpec* interpretation (Category *dInterp* object) has an associated *Spec* interpretation (Category *Interp* object) via the *Colimit*: *dInterp→Interp* functor. Thus the meaning of a diagram interpretation in terms of the class of models and reduct-functors associated with the individual *Spec* diagrams and *Spec* diagram morphisms can be defined in terms of *Spec* interpretations (Section C.3.1). Any implementation of (the colimit of) the target diagram of a *dSpec* interpretation can be extended (sometimes mechanically) to implement (the colimit of) the mediator diagram and hence (the colimit of) the source diagram.

4.3.3 Diagram interpretations compared with diagram refinement

A diagram interpretation can also be viewed as a generalization of diagram refinement (Definition C.28), [SJ95], [SLM98] although it is not used as such in this dissertation. Intuitively, since diagram refinement involves a single change in structure $G_1 \rightarrow G_2$, there must be a direct embedding of the source diagram structure in the target diagram structure. Because a diagram interpretation $G_S \rightarrow G_M \leftarrow G_T$ involves two changes of structure, based on a common mediator structure, the structure of the target diagram may bear no relationship to the structure of the source diagram. It is this additional change in shape that enables diagram interpretations to express more refinements than diagram refinement. On the other hand, a diagram refinement can be expressed as a collection of compatible interpretations [SLM98], whereas diagram interpretations are a pair of diagram morphisms, one of which is a definitional extension. Developing a diagram interpretation that takes advantage of that additional structure is more complicated.

If the mediator diagram of a diagram interpretation is restricted to be the same shape as the domain diagram and the shape morphism from the codomain diagram to the mediator diagram is restricted to be an epimorphism, then a diagram interpretation would have essentially the same "refinement" power as diagram refinement as used in practice. The definition of diagram refinement allows the target diagram to contain specifications and morphisms that are completely unrelated to the refinement operation that is taking place. In contrast, the definition of a diagram interpretation allows the mediator diagram to contain specifications and morphisms that are completely unrelated to the refinement operation that is taking place. In practice, with either definition, such additional unneeded specifications and morphisms would not be used. Thus in practice, diagram refinement is equivalent to diagram interpretations when diagram interpretations are restricted as described in Figure 4-17.

$$\begin{tikzcd}
G_S \arrow[rr, "\sigma_S"] \arrow[dd, "S"{name=S}] \arrow[ddr, "\delta_S"] & & G_M \arrow[dd, "M"'{name=M}] & & G_T \arrow[ll, "\sigma_T"'] \arrow[dd, "T"'{name=T}] \arrow[ddl, "\delta_T"'] \\
\\
\mathit{Spec} & \mathit{Spec}^{\to} \arrow[l, "\text{Source}"'] \arrow[r, "\text{Target}"] & \mathit{Spec} & \mathit{Spec}^{\to} \arrow[l, "\text{Target}"'] \arrow[r, "\text{Source}"] & \mathit{Spec}
\arrow[from=S, to=M, Rightarrow]
\arrow[from=T, to=M, Rightarrow, "d"]
\end{tikzcd}$$

If σ_S = identity morphism and σ_T = epimorphism then a diagram interpretation is roughly equivalent to a diagram refinement

Figure 4-17. Relationship between diagram interpretation and diagram refinement

Because of the lack of limits on the two definitions, the target diagram for diagram refinement vs. the mediator diagram for diagram interpretations, neither construct can emulate the other. However for the practical uses of both constructs, diagram refinements are a subset of diagram interpretations thus leading to the remark that diagram interpretations are a generalization of diagram refinement. Figure 4-18 depicts how diagram refinements that do not contain extraneous specifications and morphisms in the target diagram can be lifted to be a diagram interpretation.

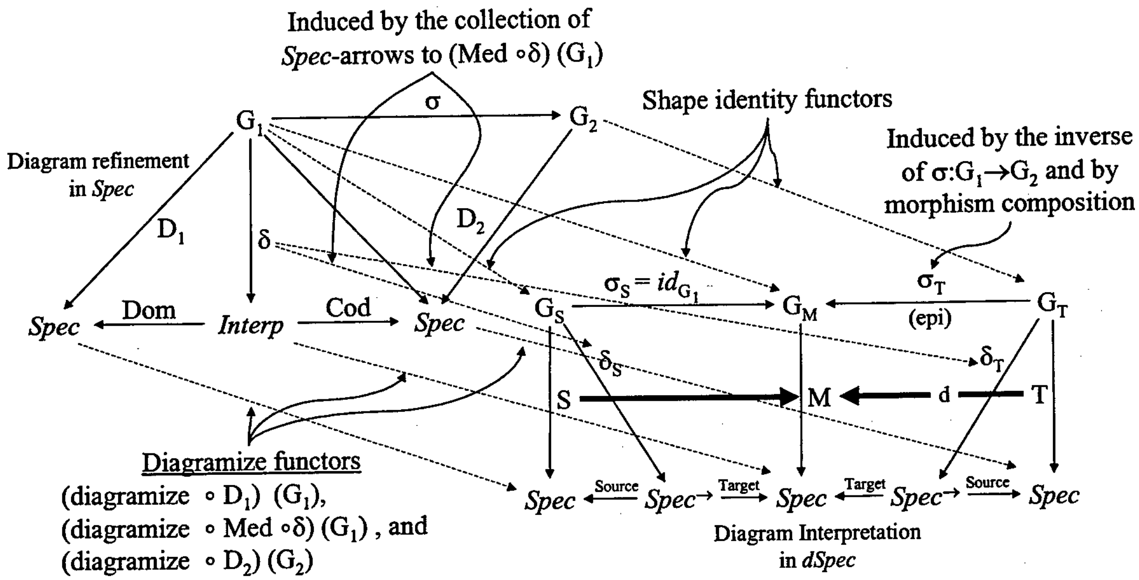


Figure 4-18. Lifting a diagram refinement to a diagram interpretation

Again, in this dissertation, diagram interpretations are used as a mechanism for representing design information and not as a refinement mechanism. The limitations of diagram refinement as a method for applying design information were presented in Section 2.5.

4.4 *Contributions and Future Work*

Although most of the groundwork for category *dSpec* was accomplished in Chapter 3, the notion of a category of diagrams of specifications is new in the literature. As shall be demonstrated in later chapters, this category enables the diagrammatic structure of a requirement specification to be refined in a manner that is superior to the current refinement technique of using diagram refinement. This category and the functor *Colimit* enable diagrams of specifications and morphisms and their diagram morphisms to be treated as specifications and morphisms in *Spec*. This makes manipulating *Spec* diagrams and diagram morphisms as simple (on a theoretical level) as manipulating specifications and specification morphisms. This also enables the model semantics of category *Spec* objects and arrows to be lifted to category *dSpec* objects and arrows.

A second contribution of this chapter is the diagram parameterization theory developed in Section 4.2, which is the focus of Chapter 5. Although many notions of algebraic specification parameterization require a conservative extension to exist between the formal parameter and the body, the notion of all of the variable parts of the body being covered by the parameter is formalized in this dissertation for the first time. Reasoning about the variable parts of a body is necessary when proving that a refinement of a body does not alter its relationship with the "true" formal parameters of a body. Also, as is discussed further in Chapter 5, the notion of the body being a diagram instead of a single specification is also a generalization of other notions of parameterization.

Of note is the notion that diagram morphisms have properties such as being a conservative extension or a definitional extension that parallels those same properties in category *Spec* arrows. These properties enable *dSpec* arrows to be treated in a similar fashion to the conservative extension (Section C.1.4) and definitional extension (Section C.1.5) *Spec* arrows. (A conservative extension enables all models of the (colimit of the) source diagram to be extended so that they are models of the (colimit of the) target diagram and a definitional extension enables such a model extension to be uniquely characterized.)

A final contribution is the notion of a *dSpec* interpretation that parallels a *Spec* interpretation and that forms the basis for representing design information in Chapter 6. While a *Spec* interpretation can insure that the content of a specification is refined, a *dSpec* interpretation can insure that the content and structure is refined.

Future work involves developing more and better ways for determining if a *dSpec* morphism (and hence a specification morphism via the colimit functor) is a conservative extension morphism or not. Also more and better ways are needed for determining if a *dSpec* object (and hence a specification via the colimit functor) has variable parts or not, i.e. determining whether a specification has non-isomorphic models.

5 Parameterized Diagrams

This chapter develops a syntax for extending, composing, parameterizing and instantiating diagrams by building upon the theory of parameterized diagrams developed in Section 3.5 and Section 4.2. Parameterized diagrams are shown to subsume other forms of algebraic specification parameterization (as presented in Section 2.3 and Appendix E).

Syntax may seem like a minor point, but it is at the heart of what enables people to think and operate in terms of a higher level abstraction. The `diagram` statement transforms a diagram construction operation into a parameter passing operation. It does not allow more types of diagrams to be created than by the nodes and arcs listing construction method. However, this paradigm shift from construction to instantiation hides unnecessary "implementation" details and enables the specifier to think about the problem being specified and not the underlying mechanism (diagrams) used to specify the problem.

This chapter is structured as follows. Section 5.1 introduces a syntax for creating, extending, and composing diagrams. Section 5.2 defines the semantics of the basic diagram statement, which can only create, combine and merge diagrams, and demonstrates its use. Section 5.3 defines the semantics of the parameterization and instantiation aspects of the diagram statement and demonstrates their use. Section 5.4 compares the diagram statement developed in this chapter with the diagram theory presented earlier.

5.1 The Diagram Statement Syntax

In this dissertation it is diagrams of specifications and morphisms that are parameterized, as opposed to other parameterization research, where it is the specifications that are parameterized. Passing parameters to the parameterized diagram (instantiation) involves extending the diagram so that, at a minimum, the missing parameterized parts of the diagram are filled in. The completed (instantiated) parameterized diagram is the structured result of passing parameters. The colimit object of the instantiated parameterized diagram is the unstructured result of passing parameters.

The diagram statement developed in this chapter enables diagrams to be extended, parameterized, instantiated, and merged. These higher-level operations make it easier for the diagram structure of a specification to be developed, and the results are more understandable that existing methods.

```
Diagram → diagram diagram-name [ [ f-param-list ] ] is
            [ import diag ( , diag )* ]
            [ nodes node ( , node )* ]
            [ arcs arc   ( , arc )* ]
            [ merge merge ( , merge )* ]
            [ instantiate ( instant )+ ]
          end-diagram

f-param-list →  ( sort-name | op-name ) ( , ( sort-name | op-name ) )*

diag →      [ diagram-name : ] diagram-ref [[ ]]

node →      [ node-name : ] spec-ref

arc →       [ arc-name : ] g-node-ref -> g-node-ref : sig-mapping-term
arc →       g-node-ref -> ? : sig-mapping-term

sig-mapping-term →     { [ sig-map-rule (, sig-map-rule )* ] }
sig-mapping-term →     morphism-ref

sig-map-rule →  ( op-ref -> op-ref ) | ( sort-ref -> sort-ref )

merge→      g-node-ref ( = g-node-ref )+
merge→      g-arc-ref ( = g-arc-ref )+

instant→  [ diagram-name : ] diagram-ref [ a-param-list ]

a-param-list → ( g-sort-ref | g-op-ref ) [ , a-param-list ]

qualified-ref → ( any-ref . )* any-ref
          (subject to semantic/type analysis restrictions, of course)
```

Key for the grammar:

Non-terminals are *underlined and italicized*

[ ] – bold square brackets mean the syntactic contents are optional.

( )* – bold starred parens mean zero or more occurrences of the contents.

( )+ – bold plussed parens mean one or more occurrences of the contents.

| – bold vertical bar means either side but not both sides (exclusive or).

Figure 5-1. Grammar for the Diagram statement

The grammar in Figure 5-1 defines the syntax of the diagram statement. The syntax for the diagram statement assumes the existence of a syntax for specifications, where a specification can be referenced by name, *spec-ref*, a specification morphism can be indicated by name, *morphism-ref*, and where the sets of sorts and operations within a specification can be referenced by name, *sort-ref* and *op-ref*.

The *-name* and *–ref* endings on a non-terminal indicate semantic information beyond the syntax defined by the grammar. To be meaningful, a diagram statement must pass the semantic analysis requirements as well.

A non-terminal ending in *-name* is syntactically a symbol but it represents a declaration of the indicated preface type. In the formal parameter list, *f-param-list*, the declaration is "split"; the symbol is known but its "type" (sort or operation) is not known until the same symbol is used elsewhere in the diagram statement.

A non-terminal ending in *-ref* may be a (qualified) symbol or an in-line declaration of the indicated type. If it is a symbol, it is required to be of the type indicated by the preface from some declaration of that symbol (The symbol associated with a *spec-ref* must be the same as one that was in a *spec-name* declaration in the enclosing scope.)

Any *-ref* non-terminal preceded by a *q-* may be a *qualified-ref*, and may be preceded by a collection of other *-refs* in order to distinguish it from other *-refs* with the same symbol name. (A *sort–ref* or *op-ref* may be qualified by *node-refs* and a *node-ref* may be qualified by *diagram-refs* in order to reference a particular object unambiguously.)

5.2 Semantics of the Basic Diagram Statement

This section defines the semantics of the basic diagram statement. The basic diagram statement does not involve parameterization or instantiation but does enable diagrams to be combined and extended . The grammar for this limited version of the diagram statement is depicted in Figure 5-2.

```
diagram →  diagram diagram-name is
             [ import diag ( , diag )* ]
             [ nodes node ( , node )* ]
             [ arcs arc   ( , arc )* ]
             [ merge merge ( , merge )* ]
           end-diagram

diag →      [ diagram-name : ] diagram-ref

node →      [ node-name : ] spec-ref

arc →       [ arc-name : ] q-node-ref -> q-node-ref : sig-mapping-term

sig-mapping-term →    { [ sig-map-rule (, sig-map-rule )* ] }
sig-mapping-term →    morphism-ref

sig-map-rule →  ( op-ref -> op-ref ) | ( sort-ref -> sort-ref )

merge→     q-node-ref ( = q-node-ref )+
merge→     q-arc-ref ( = q-arc-ref )+
```

Figure 5-2. Grammar for the basic Diagram statement (no parameterization)

5.2.1 The `nodes` clause and the `arcs` clause

The `nodes` clause indicates a collection of nodes, α_i, and their associated *Spec* objects. The `nodes` clause indicates which specifications are to be included in a diagram (or added to existing diagrams, as discussed in Section 5.2.2). Each node has the indicated name. If a name is not supplied the node has the name of the indicated specification. All nodes must have unique names.

The `arcs` clause indicates a collection of arcs, f_i, and their associated *Spec* arrows. The `arcs` clause indicates which nodes have an arc between them and the underlying specification morphism associated with the arc. Each arc has the indicated name or (if a name is not supplied) it has the generic name *node*-to-*node*.

A diagram statement that only has the `nodes` and `arcs` clauses produces the diagram *Diagram-name*:G→*Spec* as indicated by the arcs and nodes and their mapping to *Spec* objects and arrows. The shape category G is defined by the listed arcs and nodes and their orientation, Objects(G) = α_i, and Arrows(G) = f_i, along with all identity arcs and then closed under composition. The functor Diagram-name is the pair of functions < $\text{Diagram-name}_{\text{Object}}$, $\text{Diagram-name}_{\text{Arrow}}$ > as defined by the association between a node and a *Spec* object, and an arc and a *Spec* arrow. The identity arrows and the composition of two given arrows are easily determined.

The syntax and semantics of the diagram statement that contains only the `arcs` and `nodes` clauses is essentially identical to the Specware language diagram statement described in Section 2.4 and Appendix D. An example of a diagram statement that contains only an `arc` and `node` clause is diagram Map depicted in Figure 5-3.

5.2.2 The `import` clause

The `import` clause indicates a collection of *Spec* diagrams (or *dSpec* objects), D_i. Each of these diagrams are to be extended or combined into a larger diagram, thus the `import` clause enables diagrams to be constructed once and then reused and extended many times. With the `import` clause, a diagram becomes an extendable structure; it can be added to and combined with other diagrams.

The semantics of the `import` clause is defined by the Join operation (Definition 3.22), i.e. Join(D_i). If the same diagram is imported multiple times, it must be given different names. The effect of the import clause is to produce a *base diagram* of the diagram statement that may be further affected by other clauses within the diagram statement.

Every diagram that is imported has an extension morphism (Definition 3.21) to the base diagram of the diagram statement. Additional nodes (specifications) can be added to the base diagram by using the `nodes` clause. The `arcs` clause can be used to connect nodes from any of the imported diagrams or the nodes from the `nodes` clause. Should two or more nodes (that originated in different imported diagrams) have the same name, the diagram names can be used to qualify (disambiguate) the node names. In Section 5.2.1 the `nodes` and `arcs` clauses (without the `import` clause) are extending the "empty" base diagram.

The semantics of the basic diagram statement with an `import` clause is the same as without, namely a diagram Diagram-name:G→*Spec* is the result. Shape G is constructed first via the join semantics of the `import` clause and then extended with additional arcs and nodes from those associated clauses (and closed under arc identity and composition).

Figure 5-3 depicts the creation of a reusable diagram, Map, and its extension to diagram Map-from-Nat-to-Nat (as well as the colimit operation that flattens the diagram into a single specification). As can be seen in Figure 5-3, the diagram statement supports diagram extensions.

```
Diagram Map is
nodes Map,
        s1:One-Sort, s2:One-Sort
  arcs s1->Map: {X -> Dom},
       s2->Map: {X -> Cod}
end-diagram
```

```
Diagram Map-from-Nat-to-Nat is
  import Map
  nodes Dom:Empty, Cod:Empty
  arcs map.s1->Dom: {X -> Nat},
       map.s2->Cod: {X -> Nat}
end-diagram
```

```
Spec Map-from-Nat-to-Nat is
  Colimit of Map-from-Nat-to-Nat
```

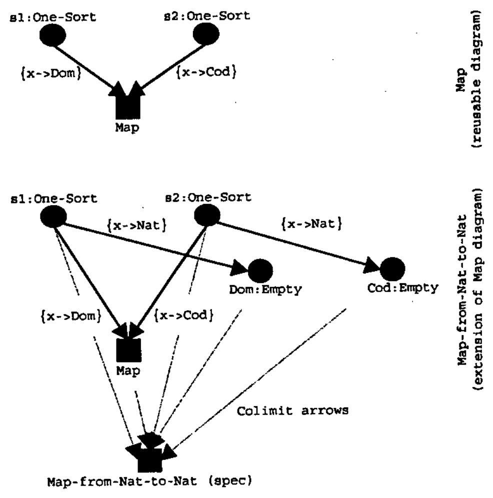


Figure 5-3. Extending an existing diagram

5.2.3 The merge clause

The merge clause enables diagram nodes (or arcs) with the identical underlying *Spec* objects (arrows) to be combined into a single node (or arc). For two or more nodes (or arcs) to be merged they must have the same underlying *Spec* object (or arrow). Given a base diagram D:G→*Spec*, formed by importing diagrams and then extending them, the merge clause with its collection of equivalenced nodes and objects induces an epimorphism σ:G→G' that satisfies the properties listed in the definition of the Fold operation (Definition 3.23). Thus Fold(D, σ) defines the semantics of the merge clause.

In Figure 5-4, the diagram statement is used to develop a new reusable diagram, Map-to-Map, by composing two individual Map diagrams (named m1 and m2 to distinguish them) so that the "codomain" object of the m1 map is also the "domain" of the m2 map. The Map-to-Map diagram is then extended via the method already described using Nat as the "domain", "mediator", and "codomain" parameters for the larger diagram. Thus the reusable library diagram Map is composed with itself to form an even larger library diagram, Map-to-Map, that is then further extended. This example was chosen not because of the usefulness of the Map-to-Map data structure per se, but because it demonstrates how the merge clause works as well as demonstrating how diagram composition can be accomplished despite the overloading of specification and node names that occurs when two copies of the same diagram are imported by one diagram statement.

```
Diagram Map-to-Map is
  Import m1:Map, m2:Map
  Merge m1.s2 = m2.s1
end-diagram
```

```
Diagram Map-to-Map-Nat is
  Import Map-to-Map
  nodes Dom:Empty, Med:Empty,
        Cod:Empty
  arcs m1.s1->Dom {X -> Nat},
       m1.s2->Med {X -> Nat},
       m2.s2->Cod {X -> Nat}
end-diagram
```

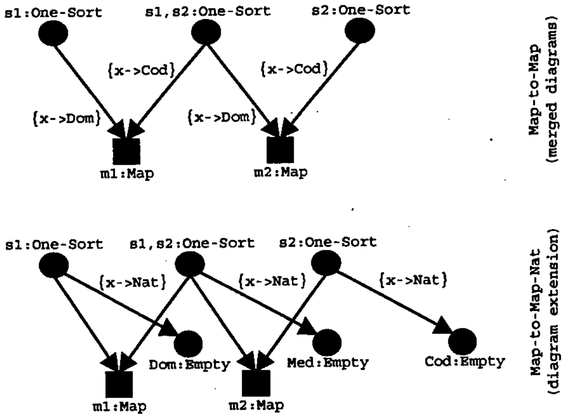


Figure 5-4. Composing (merging) diagrams

5.3 Parameterization and Instantiation Semantics of the Diagram Statement

This section describes the semantics of the diagram statement when parameterization and instantiation are used. The basic diagram statement without parameterization enables diagrams to be extended and combined. Using the basic statement by itself means that diagrams can only be constructed using a diagram building analogy (Figure 5-3 and Figure 5-4) where arcs and nodes are being added to an existing diagram.

The code in Figure 5-5 depicts two different diagram constructions for a Set of items: a set of Nats and a set of Flags. Note that the code for the two diagram statements is identical except for the nodes (specifications) Empty and Flag and the sorts Nat and Flag as indicated by the underlining in Figure 5-5. Note also that since the source of the morphism is specification One-Sort in both cases, half of the code for the morphism is already known, namely "X - >". Given a target sort, which has been identified as coming from a particular specification, it ought to be possible to fill in the underlined pieces in the code in Figure 5-5. This observation forms the basis for the parameterization and instantiation syntax for diagrams.

```
diagram Set-of-Nat is
  Nodes Set, One-Sort, Empty
  Arcs One-Sort -> Set:    {X->E}
       One-Sort -> Empty: {X->Nat}
end-diagram
```

```
diagram Set-of-Flag is
  Nodes Set, One-Sort, Flag
  Arcs One-Sort -> Set:   {X->E}
       One-Sort -> Flag: {X->Flag}
end-diagram
```

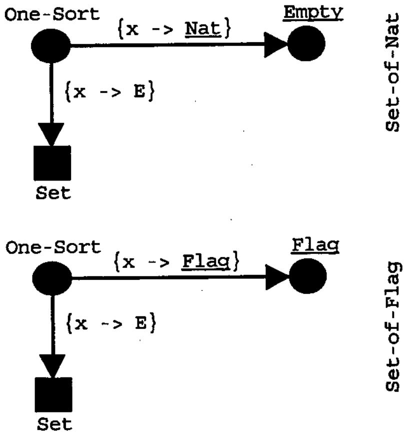


Figure 5-5. Diagram similarities using the diagram construction method

5.3.1 Formal parameters

The formal parameters of a parameterized diagram are signature elements, not diagrams, specifications, or nodes and arcs (although the signature elements are representative of the required nodes and arcs). Figure 5-6 contains the grammar for the parameterized version of the diagram statement. Note that the formal parameter list is made up of sort names and operation names. Note also that the `arc` clause has been extended to include a new/different "right hand side" that has an anonymous arc (no arc name and no specified target node). This anonymous arc is used to define the "type" (sort or operation) of the formal parameter signature elements as well as whether the signature elements must come from the same specification or not. The anonymous arc can only be instantiated as described in the sequel; it cannot be merged (as it has no name to reference it).

```
diagram → diagram diagram-name [ [ f-param-list ] ] is
             [ import diag (, diag )* ]
             [ nodes node (, node )* ]
             [ arcs arc   (, arc )* ]
             [ instantiate ( instant )+ ]
          end-diagram

f-param-list →  ( sort-name | op-name ) [ , f-param-list ]

diag →     [ diagram-name : ] diagram-ref [[ ]]

arc →      q-node-ref -> ? : sig-mapping-term

sig-mapping-term →    { [ sig-map-rule (, sig-map-rule )* ] }
sig-mapping-term →    morphism-ref

sig-map-rule →  ( op-ref -> op-ref ) | ( sort-ref -> sort-ref )

instant→  [ diagram-name : ] diagram-ref [ a-param-list ]

a-param-list → ( q-sort-ref | q-op-ref ) [ , a-param-list ]
```

Figure 5-6. Grammar for the parameterized Diagram statement

The parameter list associated with a diagram statement enables a diagram to be parameterized by sorts and operations that can be used to complete a signature mapping between specifications. For each sort or operation in the parameter list there must be an associated by-name reference in an anonymous arc declaration of the following form:

```
arc → node-ref -> ? : sig-mapping-term.
```

In an anonymous arc, the source node is known but the target node and morphism to the target node are not known. Since the source node is known, however, the source sorts and operations in the underlying signature mapping are known as well, thus the source signature references on the left of the "->" symbol in the following syntax are known,

```
sig-map-rule → ( op-ref -> op-ref ) | ( sort-ref -> sort-ref ),
```

but the target signature references on the right are not known. Each target reference in the signature mapping of an anonymous arc must match up by name with a formal parameter list item. In this case the type of the source reference (sort or op) indicates the type of the target reference and the type of the associated formal parameter list item.

The formal parameters, a collection of sort and operation names, represent missing *Spec* arrows and objects. The known nodes, arcs, and imported diagrams of a parameterized diagram statement produce a diagram, *diagram-name*:G→*Spec*, the same as the non-parameterized version above did. However, shape G along with the unknown arcs and target nodes (based on the parameterization) also define a shape G' and a shape monomorphism σ:G→G'. Thus the semantics of a diagram statement with formal parameters is a parameterized diagram, < *diagram-name*:G→*Spec*, σ:G→G' > (Definition 3.24). The diagram *diagram-name*:G→*Spec* is referred to as the underlying diagram of the parameterized diagram or the fixed part of the parameterized diagram. The shape monomorphism σ indicates what additional arcs and nodes (and associated *Spec* objects and arrows) are needed to instantiate the diagram.

Figure 5-7 depicts a diagram parameterization involving the specification Set. The semantics of the code in Figure 5-7 is the parameterized diagram, <Set:G→*Spec*, σ>, where diagram Set is the diagram with a single arrow from specification One-Sort to specification Set, and σ:G→G' is the shape monomorphism that extends shape G with an additional node, and an arc to that node as depicted on the right in Figure 5-7.

```
diagram Set [Elem] is
  Nodes Set, One-Sort
  Arcs One-Sort -> Set: {X->E}
       One-Sort -> ?: {X -> Elem}
end-diagram
```

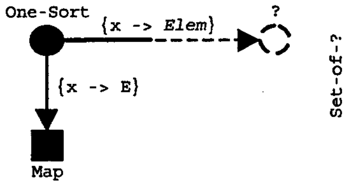


Figure 5-7. Formal parameter example

Figure 5-8 depicts an example of diagram parameterization with multiple parameterized objects. Note that the names dom and cod are multiply defined in the parameterization. In the first two arcs, dom and cod are sorts within the (target) Map specification. In the last two (anonymous) arcs, dom and cod are placeholders for the actual signature elements as the target specification is not yet known. Duplicate names such as these easily coexist in the same diagram statement.

```
Diagram Map [dom, cod] is
Nodes Map,
      s1:One-Sort, s2:One-Sort
Arcs s1 -> Map: {X->dom}
     s2 -> Map: {X->cod}
     s1 -> ?: {X->dom}
     s2 -> ?: {X->cod}
end-diagram
```

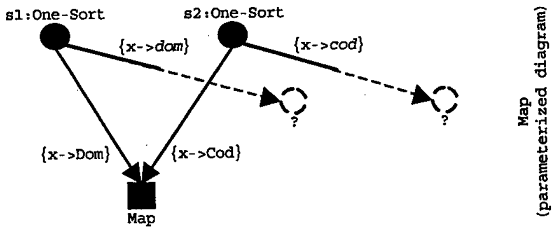


Figure 5-8. Map parameterized diagram example

5.3.2 The instantiate clause

In order to instantiate a parameterized diagram one must first import the parameterized diagram. The import clause permits parameterized diagrams, indicated by the empty brackets, [ ], as well as normal diagrams to be imported. The semantics of the import clause have not been changed from the discussion in Section 5.2.2. When a diagram, including the "fixed" diagram underlying a parameterized diagram, is imported it becomes part of the diagram being produced by the diagram statement by the join semantics (Definition 3.22). Each imported parameterized diagram must be instantiated using the instantiate clause. (Leaving the brackets off of an imported parameterized diagram indicates the fixed portion of the parameterized diagram is imported, and no instantiation is needed.)

In the instantiate clause, actual sorts and operations are passed to the parameterized diagram via the sequence former "[ ]". By qualifying the passed signature elements until their parent specifications (and nodes) can be unambiguously determined, the signature elements carry with them the target specification (and node) information for the anonymous arc being instantiated. All the signature elements associated with one of the unknown arcs within the parameterized diagram being instantiated must come from the same specification associated with the same node. If they do, the passed signature elements themselves complete the signature mapping in the morphism associated with this arc and the node they come from becomes the target node of that mapping. The instantiated arc is named in the generic way and the unknown target node of the arc assumes the name of the "indirectly" passed node.

Because the imported diagrams and nodes and arcs were present prior to instantiation (so to speak) in the semantics of the diagram statement, instantiation can be viewed as forming arcs between existing nodes in the base diagram. If the instantiation is named, then the nodes and arcs of that sub-diagram can be referenced by the instantiation name.

The key to being able to use signature elements as parameters is to ensure that the signature elements are unambiguously attributed (sourced) to a single node in the diagram. It may be possible to construct specifications and diagrams such that two signature elements within a single specification have the same name and cannot be distinguished. The use of node names during construction for specifications and diagrams can eliminate such an occurrence. Given that the parameterized diagrams and specifications used by an `instantiate` statement have been properly constructed, then the signature elements within them can be unambiguously sourced when instantiating the parameterized diagram.

The result of a diagram statement with an `instantiate` clause is a diagram. That portion of the overall diagram that corresponds to a strict instantiation of the parameterized diagram is a sub-diagram of the overall diagram. Figure 5-9 depicts three different example instantiations of the Map parameterized diagram. In the first example, the Map parameterized diagram is instantiated using two different nodes/specifications. In the second example, the Map parameterized diagram is instantiated using a single node/specification. In the last example, the Map parameterized diagram is instantiated using two different nodes with the same underlying specification.

```
diagram Map-Nat-to-Flag is
  import Map[]
  nodes Empty, Flag
  instantiate
    Map[Empty.nat, flag]
end-diagram
```

```
diagram Map-Flag-to-Flag-1 is
  import Map[]
  node Flag
  instantiate
    Map[flag, flag]
end-diagram
```

```
Diagram Map-Flag-to-Flag-2 is
  import Map[ ]
  node f1:Flag, f2:Flag,
  instantiate
    Map[f1.flag, f2.flag]
end-diagram
```

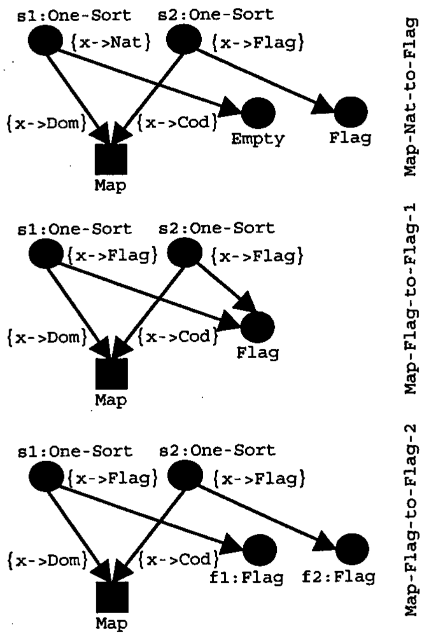


Figure 5-9. Three instantiations of the Map parameterized diagram

5.3.3 Nested parameter passing semantics

The instantiations in the examples so far have been of a parameterized diagram being instantiated by one or more nodes. Nested parameter passing enables one parameterized diagram to be instantiated by another (parameterized) diagram. This enables parameterization and instantiation to be used for constructing diagrams representing complex nested data structures. Figure 5-10 depicts a diagram of a map from a nat to a set of flags constructed using only specifications and morphisms. Note how having specifications and morphisms as the largest building blocks makes constructing even a simple diagram difficult.

```
Diagram Map-Nat-to-Set-of-Flag is
  Nodes Map, Set, Flag, Empty,
        s1:One-Sort, s2:One-Sort,
        s3:One-Sort
  Arcs s1 -> Map:    {X->Dom}
       s1 -> Empty:  {X->Nat}
       s2 -> Map:    {X->Cod}
       s2 -> Set:    {X->Set}
       s3 -> Set:    {X->E}
       s3 -> Flag:   {X->Flag}
end-diagram
```

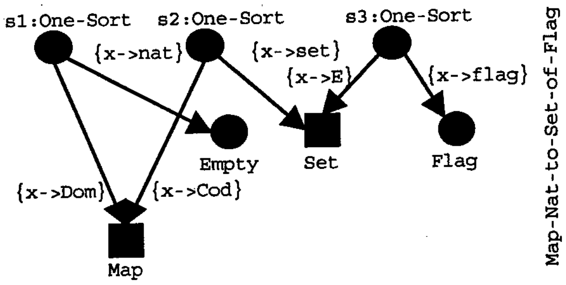


Figure 5-10. Map to Set diagram using the diagram construction method

In contrast, the same Map-Nat-to-Set-of-Flag diagram in Figure 5-10 is constructed in two distinct ways in Figure 5-11. In the first example, the diagram is constructed with two separate instantiations. The second instantiation is to the sort Set is specification Set in the (now instantiated) parameterized diagram Set. In the second example, the Set instantiation is done in-line. Comparing the examples in Figure 5-11 with the diagram statement of Figure 5-10 shows how parameterization and instantiation make it easier to understand what data structure is being constructed.

```
diagram Map-Nat-to-Set-of-Flag-1 is
  import Map[], Set[]
  nodes Empty, Flag
  instantiate
    Set[Flag]
    Map[Empty.nat, Set]
end-diagram
```

```
diagram Map-Nat-to-Set-of-Flag-2 is
  import Map[], Set[], Empty, Flag
  instantiate
    Map[Empty.nat, Set[Flag].Set]
end-diagram
```

Figure 5-11. Map to Set diagram using nested instantiation

5.3.4 Partial instantiation semantics

Instantiating a parameterized diagram is not an all-or-nothing proposition as it is with notions of parameterized specifications in the literature. A parameterized specification can be partially instantiated by passing to it formal parameters of the enclosing parameterized diagram. A new "larger" parameterized diagram is created by partially instantiating one or more other parameterized diagrams. The formal parameters of the enclosing parameterized diagram are defined either by anonymous arcs (as described in Section 5.3.1) or they are defined by passing through the "type information" of the formal parameters of a parameterized specification that is being partially instantiated. An obvious requirement for signature element parameters is that if there are a number of formal sorts and operations associated with a single unknown arc, then either all or none of them must be instantiated with actual sorts and operations, or passed up as parameters to the next level.

In Figure 5-12, the Map parameterized diagram is partially instantiated with the fixed part of the Set parameterized diagram. The Set parameterized diagram is not instantiated at all. The formal parameters of the Map-to-Set parameterized diagram are defined based on the parameters of the Map and Set parameterized diagrams that were not instantiated.

```
diagram Map-to-Set[A,B] is
  import Map[], Set[]
  instantiate
    Map[A, Set[B].Set]
end-diagram
```

```
diagram Map-Nat-to-Set-of-Flag-3 is
  import Map-to-Set[]
  nodes Empty, Flag
  instantiate
    Map-to-Set[Empty.nat, Flag]
end-diagram
```

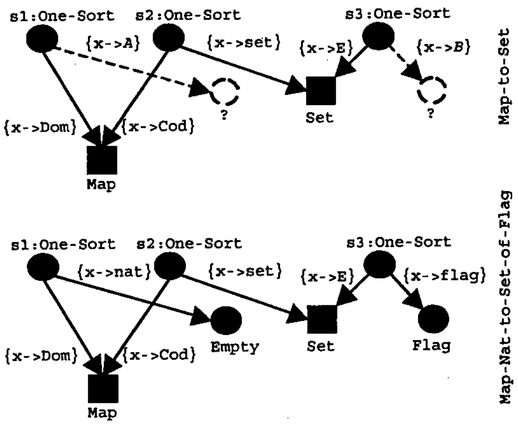


Figure 5-12. Partial instantiation example

5.3.5 Recursive parameter passing semantics

Recursive parameter passing involves instantiating two or more parameterized diagrams with each other's "body" specifications. Recursive parameter passing is needed in order to construct a recursive data structure.

The example depicted in Figure 5-13 develops a recursive data structure using the diagram construction method. The coproduct and pair (tuple) specifications may be built into the language as a primitive (they are in the Specware language). However, the code for them is included in Figure 5-13 in order to make the example easier to understand. The specifications in Figure 5-13 can be considered to be reusable library specifications while the diagram statement cannot be, as it has already been "instantiated". The colimit specification of the Recursive-Pairs-of-Nats diagram contains a sort Cop and a sort Pair. The sort Cop is a coproduct that can be the sort Pair or the sort Nil. The sort Pair is a tuple that contains a Nat and a Cop. If <, > is used to designate the elements of a pair then "`<6, <2, <1, nil>>>`" is an example of an "element" of the recursive data structure formed in Figure 5-13.

```
spec coproduct is
  sort cop, A, B, A-cop, B-cop
  op embed-A: A -> cop
  op embed-B: B -> cop
  op is-A: cop -> boolean
  op is-B: cop -> boolean
  op get-A: A-cop -> A
  op get-B: B-cop -> B
constructors {embed-A, embed-B}
                   construct cop
sort-axiom A-cop = cop | is-A
sort-axiom B-cop = cop | is-B

  Forall a:A, b:B
  ax get-A(↓(embed-A(a))) = a
  ax get-B(↓(embed-B(b))) = b
end-spec
```

```
spec pair is
  sort pair, A, B
  op make-pair: A,B -> pair
  op Project-A: pair -> A
  op Project-B: pair -> B
  constructors {make-pair}
                 construct pair
  Forall a:A, b:B
  ax project-A(make-pair(a,b)) = a
  ax project-B(make-pair(a,b)) = b
end-spec
```

```
spec nil is
  sort nil
  op nil: -> nil
constructors {nil} construct nil
end-spec
```

```
Diagram Recursive-Pairs-of-Nats is
  nodes cop:coproduct,
        pair, nil, Empty,
        T1:One-Sort, T2:One-Sort,
        P1:One-Sort, P2:One-Sort
  arcs T1->cop:    {X->A},
       T2->cop:    {X->B),
       T1->nil:    {X->nil)
       T2->pair:   {X->pair},
       P1->pair:   {X -> A},
       P2->pair:   {X-> B},
       P1->Empty:  {X-> Nat}
       P2->cop:    {X-> cop},
end-diagram
```

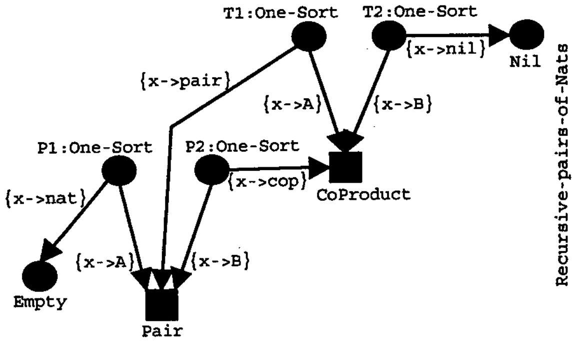


Figure 5-13. Recursive data structure using the diagram construction method

The same recursive data structure can be coded using parameterized diagrams and the `instantiate` statement in a more understandable fashion. In Figure 5-14 the parameterized diagrams for a coproduct and a Pair are constructed. Then the parameterized diagram Recursive-Pair is constructed in a way that leaves the actual "data type" portion of the abstract data structure as a parameter. These three diagrams can be considered to be reusable parameterized diagrams that could be placed in a library. Finally, the parameterized diagram Recursive-Pair is instantiated with Nats so that the resulting diagram matches that of Figure 5-13. Once again, when the method of construction with reusable parameterized diagrams (Figure 5-14) is compared to the construction method without parameterized diagrams (Figure 5-13), it is easy to see how parameterized diagrams and the diagram statement enable a specifier to work on a higher level of abstraction.

```
diagram coproduct [A,B]is
  nodes cop:coproduct,
        T1:One-Sort, T2:One-sort
  arcs T1->cop:   {X->A},
       T2->cop:   {X->B),
       T1->?:  {X->A},
       T2->?:  {X->B}
End-diagram
```

```
diagram pair [A,B] is
  nodes pair,
        P1:One-Sort, P2:One-Sort
  arcs P1->pair:  {X-> A},
       P2->pair:  {X-> B},
       P1->?:  {X-> A},
       P2->?:  {x-> B}
end-diagram
```

```
Diagram Recursive-pairs[Ele] is
  Import coproduct[], pair[]
  nodes nil
  Instantiate
    coproduct[nil,pair]
    pair[Ele,cop]
end-diagram
```

```
Diagram Recursive-Pairs-of-Nats is
  import Recursive-pairs[]
  nodes Empty
  instantiate
      Recursive-pairs[Empty.Nat]
end-diagram
```

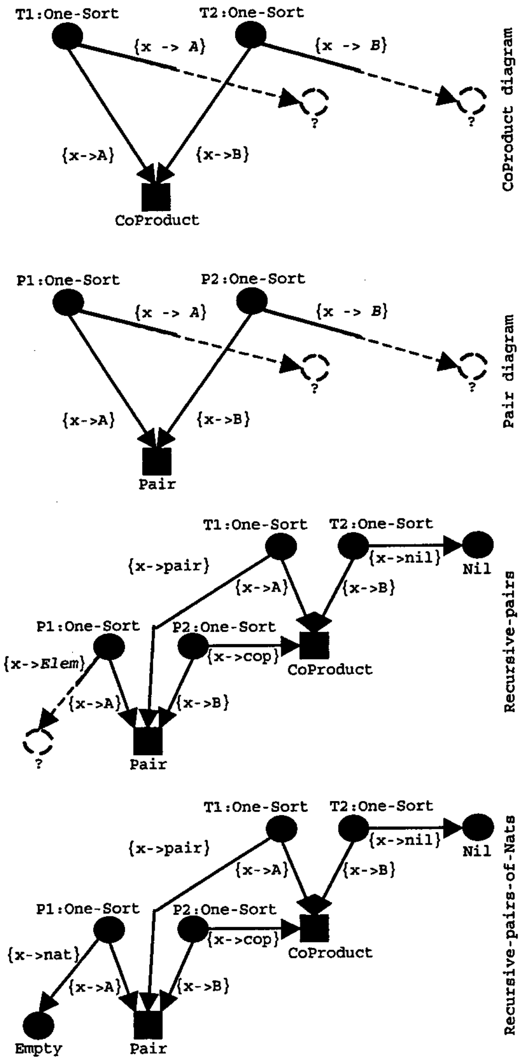


Figure 5-14. Recursive data structure using diagram instantiation

5.3.6 Syntactic sugar and Petri Net example

In a sense the `import` clause is redundant for importing parameterized diagrams (but not for non-parameterized diagrams). As each imported parameterized diagram must be instantiated within the diagram statement, the "imported" parameterized diagrams can be determined by the contents of the `instantiate` clause alone. The only requirement is that if the same parameterized diagram is instantiated more than once, then each instantiation must be given a unique name. This provides the equivalent semantics to importing the diagram multiple times in the `import` clause and naming it differently each time. Figure 5-15 uses these alternate semantics to develop the abstract requirements for a Petri Net data structure. (The code in Figure 5-15 assumes that parameterized diagrams for Set, Bag, Pair and Map already exist, which is reasonable considering their utility.) It is straightforward to see how using parameterized diagrams makes it easier to construct large complex objects and to understand what has been constructed (compare Figure 5-15 with Appendix D).

```
Diagram Petri-Net is
  nodes Empty, Place, Transition
  instantiate
    Set-of-Places:          Set[Place]
    Set-of-Transitions:     Set[Transition]
    Input-Arc:              Pair[Place, Transition]
    Output-Arc:             Pair[Transition, Place]
    Bag-of-Input-Arcs:      Bag[Input-Arc.Pair]
    Bag-of-Output-Arcs:     Bag[Output-Arc.Pair]
    Map-of-Markings:        Map[Place, Empty.Nat]
end-diagram
```

Figure 5-15. Petri Net diagram created using parameterized diagrams

5.4 Diagram Statement Compared with Parameterized Diagram Theory

How does the syntax and semantics of the diagram statement developed in this chapter compare with the diagram theory presented in Chapter 3 and Chapter 4? The theory requires that there be a formal parameter part and a body part with a diagram morphism between them, yet the diagram statement appears to make no such distinction.

The diagram statement defined in this chapter only allows parameterized diagrams with lower bound requirements (Section 3.5.7 and Section 4.2). As such, the anonymous arcs can be used to induce a separation of the fixed part of the parameterized diagram into a formal parameter part and a body part (Section 4.2.3, Figure 4-11). The formal parameter part is the source nodes of the anonymous arcs (and the arcs, if any, between these nodes) that serve as the lower bound requirement for the variable part. The body consists of all of the rest of the nodes and the arcs between them. Connecting the formal parameter part and the body should be a collection of morphisms that can be used to induce a diagram morphism between the formal parameter part and the body part. Instantiating the parameterized diagram can also induce a diagram morphism between the formal parameter diagram and the (induced) actual parameter diagram.

If one examines the various parameterized diagrams developed in this chapter, in each case it is easy to see how this partition (Definition 3.20) of the parameterized diagram can be accomplished. For example, the parameterized diagrams in Figure 5-14 can be partitioned as depicted in Figure 5-16.

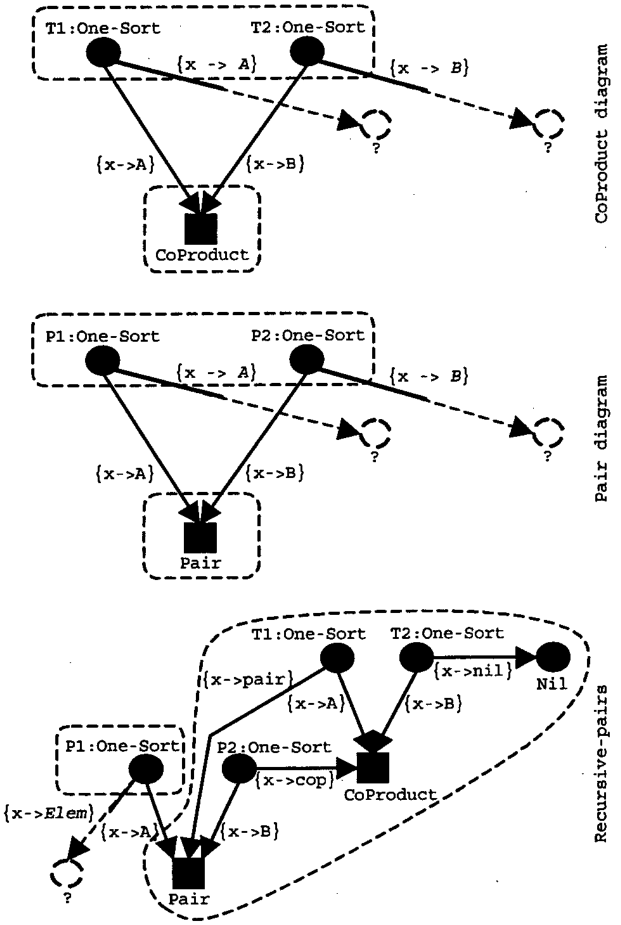


Figure 5-16. Induced partitioning of a parameterized diagram

This partitioning of the parameterized diagram into a formal parameter part and a body part cannot always be induced for reasons discussed in Section 3.4.4. However, whether or not such a partition is possible can be checked syntactically via the partition construction algorithm presented in Section 3.4.4. The existence of the diagram morphisms from the formal parameter diagram to the body diagram (and from the formal parameter diagram to the actual parameter diagram) can be enforced as a requirement of the diagram statement via semantic analysis if desired.

5.5 Contributions

The contributions of this chapter include the use of *Spec* diagrams as reusable extendable objects in a software-engineering context. The syntax for creating and instantiating parameterized diagrams bridges the gap between the underlying theory and its use in a software-engineering environment. The syntax of the diagram statement is defined in terms of the parameterized diagram theory developed in Section 3.5 and Section 4.2 and is demonstrated to be capable of single parameterization, multi-parameterization and diagram parameterization as well as nested and recursive instantiation. In addition, the parameterized diagram statement is demonstrated to be capable of being partially instantiated so that larger parameterized diagrams can be constructed from smaller parameterized diagrams.

No longer are specifications, morphisms and interpretations the largest reusable objects for creating *Spec* diagrams. The use of parameterized *Spec* diagrams enables *Spec* diagrams to be created using instantiation, which is arguably a more natural and higher-level diagram development method than the diagram construction method of listing the diagrams nodes and arcs. While previous research in parameterization of algebraic specifications has focused on how a single specification can be parameterized by a variety of different formal parameter schemes (presented in Section 2.3 and Appendix E), diagram parameterization has been shown to subsume these other notions of algebraic specification parameterization while adding additional capability. Specifically, parameterized diagrams enable the body of a parameterized object to be a diagram of specifications and morphisms instead of a single specification and the instantiation semantics result in a diagram instead of a single specification.

6 Representing and Applying Design Information

In order to refine a requirement specification that has structure, meaning it was developed by combining many smaller specifications, one has to be able to deal directly with that structure. Diagram interpretations provide the foundation for representing and applying structured design information. This chapter combines the concept of parameterized diagrams developed in Section 4.2 with the notion of diagram interpretations developed in Section 4.3, enabling the refinement of parameterized diagrams to be correctly represented. A representation method for design information is incomplete without an application method. This chapter also adapts the *dSpec* pushout operation developed in Section 3.6 in order to apply the structured design information. The representation technique for design information and its application method are shown to be able to correctly represent the refinements of parameterized diagrams and to be able to correctly apply that information to the structure of a requirement specification. The design information representation technique and application method are shown to solve the problems with using *Spec* interpretations and diagram refinement as the means for representing and applying design information, respectively, as presented in Section 2.5

This chapter is structured as follows. Section 6.1 defines how design information can be represented using diagram interpretations that have several additional requirements placed on them depending on the class of design information being represented. Section 6.2 defines how design information can be applied to requirement specifications as well as how it can be composed with other design information. Section 6.3 describes some of the ways that the representation of design information can be simplified and still be correctly applied and composed. Section 6.4 provides an example of a requirement diagram being refined by applying a sequence of design decisions. Diagram interpretations are shown to subsume interpretations and diagram refinement by solving the issues presented in Section 2.5 and offering greater assurance that design information is being represented and applied correctly.

6.1 Representing Design Information

In diagram refinement, design information is represented as *Spec* interpretations and the refinement of a diagram is based on a collection of compatible *Spec* interpretations from the specifications making up the diagram. Thus the structure of the requirement specification pre-determines the structure of the class of possible implementations. In order to break the structure dependencies inherent in diagram refinement, an interpretation mechanism is needed that can refine diagrams of specifications in such a way that a compatible family of (parallel) interpretations is not required.

A diagram interpretation can be viewed as a refinement mechanism that demonstrates how a particular *Spec* diagram can be extended in order to implement another *Spec* diagram. This refinement is independent of the structures of the source and target diagrams, as the mediator diagram intercedes. A diagram interpretation can also be viewed as design information that can be applied to a requirement specification. Unfortunately the definition of a diagram interpretation given in Definition 4.7 is insufficient to ensure that design information can be adequately represented and applied. Specifically, it has no notion of parameterization, which should be an integral part of the construction and refinement of diagrams that were created via parameterization and instantiation. In order to rectify that lapse, three different classes of design information are defined in this section along with the additional properties that are required of a diagram interpretation to insure that such design information is correctly represented and applied.

6.1.1 Classes of design information

The three different classes of design information defined in this section are *simple design information*, *parameterized design information*, and *constrained parameterized design information*. All of them require additional properties above and beyond those given in the definition of a diagram interpretation in order for diagram interpretations to represent and apply the design information correctly. The additional properties needed to represent these classes of design information are presented in this section. Section 6.1.2 provides examples of these classes of diagram interpretations.

Simple design information

Simple design information is characterized by only having a single model (class of isomorphic models) associated with it. This class of design information is represented by a diagram interpretation where the domain diagram of the interpretation has no variable parts. An example of this is the interpretation Flag→FlagAsNat←d–Nat. The additional property required by a simple diagram interpretation is that the mediator and codomain diagrams of the diagram interpretation must also have no variable parts. A diagram interpretation with these properties is termed a simple diagram interpretation. Such a diagram interpretation ensures that once a single model has been established, one cannot introduce extraneous "variability".

Parameterized design information

Parameterized design information is characterized by being associated with a parameterized diagram. This class of design information is represented by a diagram interpretation where the domain, mediator and codomain diagram making up the diagram interpretation can be induced to have a body part and a formal parameter part. An example of this is the diagram interpretation Set→SetAsBag←d–Bag in Figure 4-14. The following properties (which are also depicted in Figure 6-1) are required by parameterized diagram interpretations:

1. The entire diagram interpretation must be capable of being (re)partitioned so that the domain, mediator and codomain diagrams have separate formal parameter and body diagrams.
2. In the (re)partition, the formal parameter diagrams form a diagram interpretation and the body diagrams form a diagram interpretation and there exists a (conservative) diagram interpretation morphism between the two.
3. The diagram morphism from the domain formal parameter diagram to the mediator formal parameter diagram must be a definitional extension morphism (typically the identity morphism).

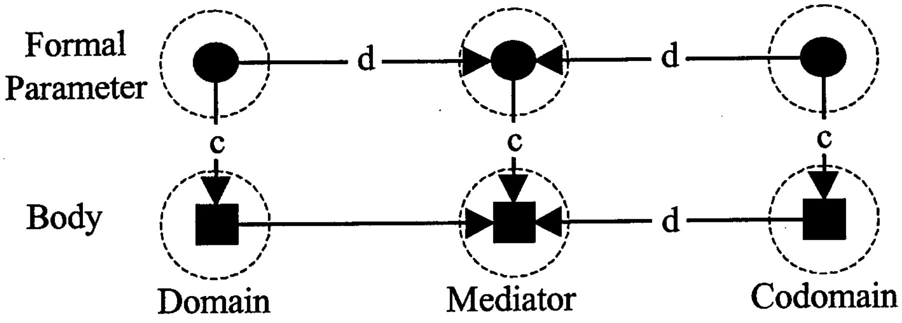


Figure 6-1. Required properties of a parameterized diagram interpretation

Property 1 ensures that the parameterized nature of the domain diagram is carried through to the mediator and target diagrams. Property 2 ensures that the body diagram is being refined in a way that is compatible with the way the formal parameter diagram is being refined. The refined formal parameter covers the refined body in the same way. Property 3 ensures that the formal parameter diagram is (in essence) not being refined at all as all formal parameter morphisms are definitional extension morphisms (typically the identity morphism). A parameterized diagram interpretation ensures that no additional constraints are added to the parameterized part (and no additional variability is added to the parameterized part) of the mediator and codomain diagrams.

As an example, this last property ensures that the formal parameter One-Sort in the domain diagram will not turn into the formal parameter Total Order in the mediator diagram as not every instantiation of One-Sort will have a total order operation. Together the three properties ensure that it is only the body part that is being refined by the diagram interpretation and not the formal parameter part.

Constrained parameterized diagram interpretation

Constrained parameterized design information is characterized by being associated with a parameterized diagram that has been instantiated with some constraint diagram. This class of design information is represented by a constrained parameterized diagram interpretation where the domain diagram can be induced to have a body part, a formal parameter part, and a constraint part. (Which is an instantiation of the formal parameter part that further restricts the variable part of the body.) An example of where such a diagram interpretation is needed is refining a Set-of-*something* to a Tree-of-*something* because the *something* must have a Total-Order operation defined over it. The following properties (which are depicted in Figure 6-2) are required by constrained parameterized diagram interpretations:

1. The entire diagram interpretation must be capable of being (re) partitioned so that the domain, mediator and codomain diagrams have separate formal parameter and body parts and that the domain and mediator diagrams have an "actual parameter" part in addition to the formal parameter and body parts that represents the "constraint".
2. In the (re)partition, the formal parameter diagrams form a diagram interpretation and the body diagrams form a diagram interpretation and there exists a (conservative) diagram interpretation morphism between the two.
3. The diagram morphism from the domain constraint diagram to the mediator constraint diagram must be a definitional extension morphism (typically the identity morphism).
4. The diagram morphism between the mediator formal parameter diagram and the mediator constraint diagram must be a definitional extension morphism (typically the identity morphism).

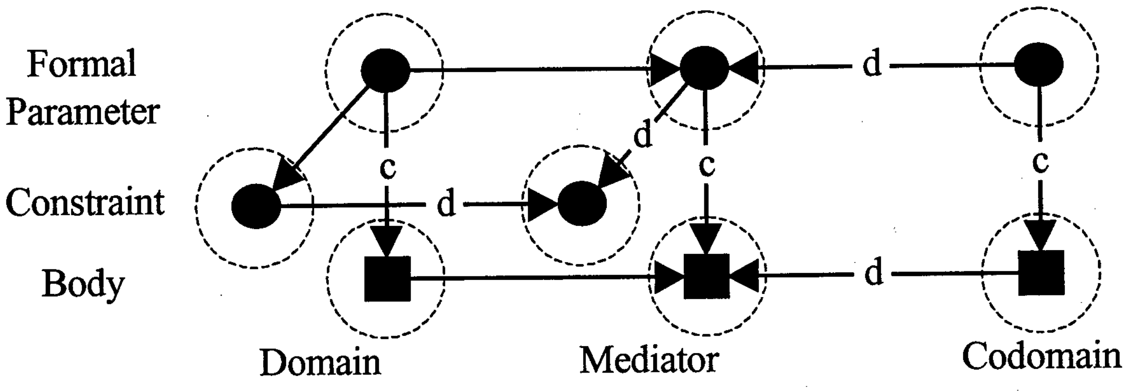


Figure 6-2. Required properties of a constrained parameterized diagram interpretation

Properties 1 and 2 have purposes similar to the related required properties of parameterized diagram interpretations. Properties 3 and 4 ensure that the additional restrictions imposed on the formal parameter part, as indicated by the constraint diagram, are embedded in the formal parameter part of the mediator and target parameterized diagrams. Thus if an actual parameter meets the requirements in the constraint diagràm (as well as the requirements in the formal parameter diagram) then the refinement as a whole is possible.

6.1.2 Examples of design information

A parameterized diagram interpretation Bag ⇒ List is depicted in Figure 6-3. The Bag⇒List diagram interpretation satisfies the required properties of a parameterized diagram interpretation as it can be partitioned as depicted in Figure 6-4.

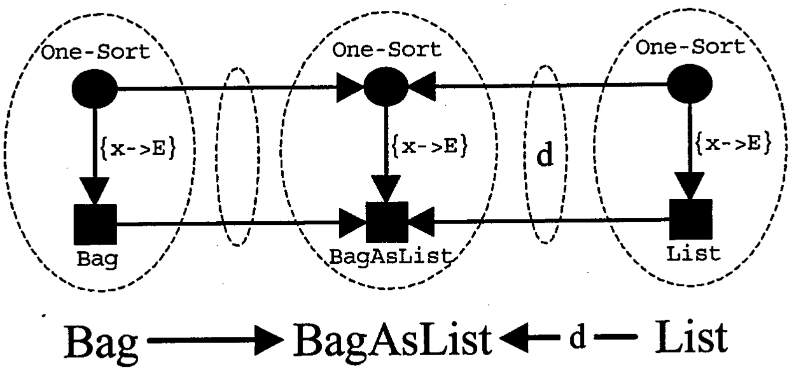


Figure 6-3. Bag as List diagram interpretation

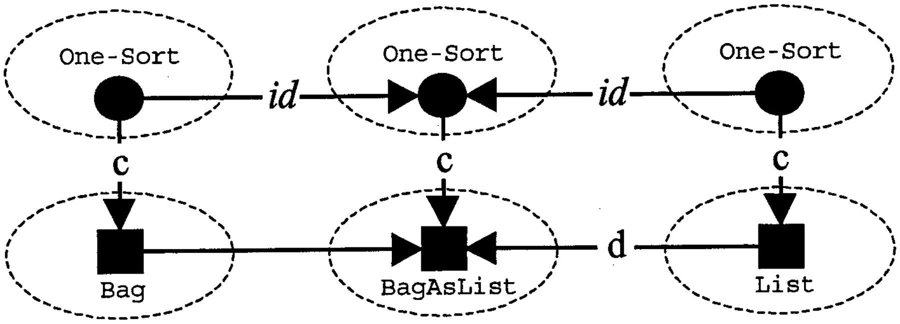


Figure 6-4. Bag as List parameterized diagram interpretation

A more complex example of a parameterized diagram interpretation is depicted in Figure 6-5, where a Bag of Pair is interpreted as a Map to a Bag. Note that the structure of the domain body is completely different from the structure of the codomain body. There is no one-to-one mapping between the domain body specifications and the target body specifications. The code for the diagram interpretation is in Appendix F, and the partition that demonstrates that it satisfies the required properties of a parameterized diagram interpretation is depicted Figure 6-6.

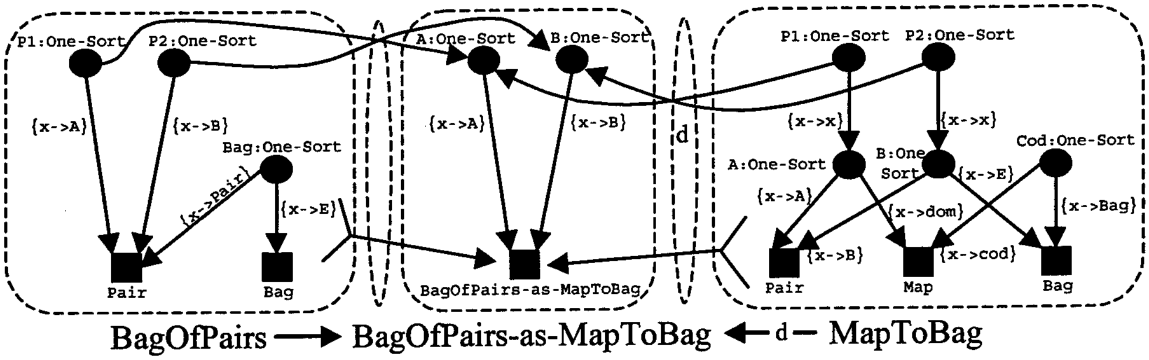


Figure 6-5. BagOfPairs-as-MapToBag diagram interpretation

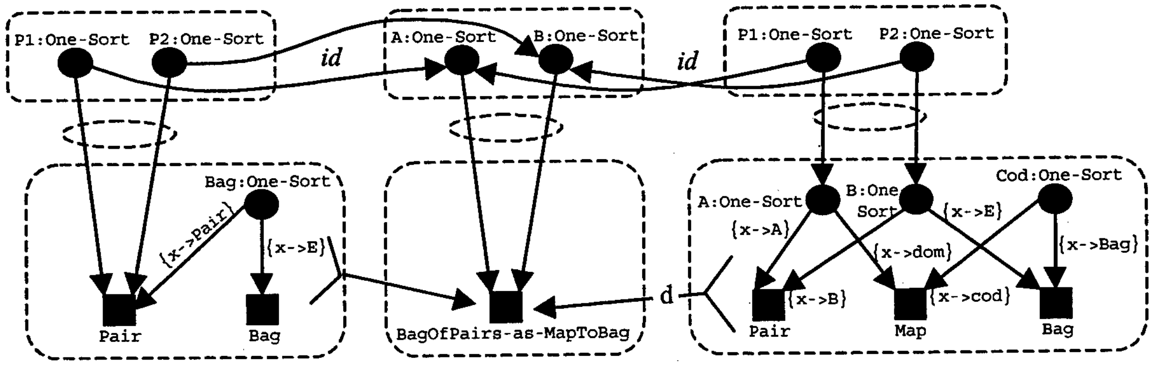


Figure 6-6. BagOfPairs-as-MapToBag parameterized diagram interpretation

Figure 6-7 depicts a constrained parameterized diagram interpretation. The domain diagram of the diagram interpretation has a Set specification for the body, the One-Sort specification as the formal parameter and the Total-Order specification as the constraint. The meaning of the Set-TO ⇒ Tree diagram interpretation is that any Set that has been instantiated with a sort that has a total order defined over it can be refined into a Tree where the elements are stored in sorted order. The code for Figure 6-7 is in Appendix F and the partition that demonstrates that it is a valid constrained parameterized diagram interpretation is in Figure 6-8.

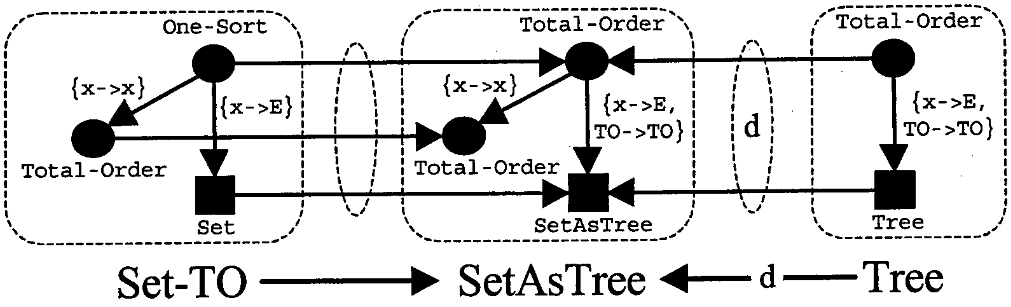


Figure 6-7. Set-TotalOrder-as-Tree diagram interpretation

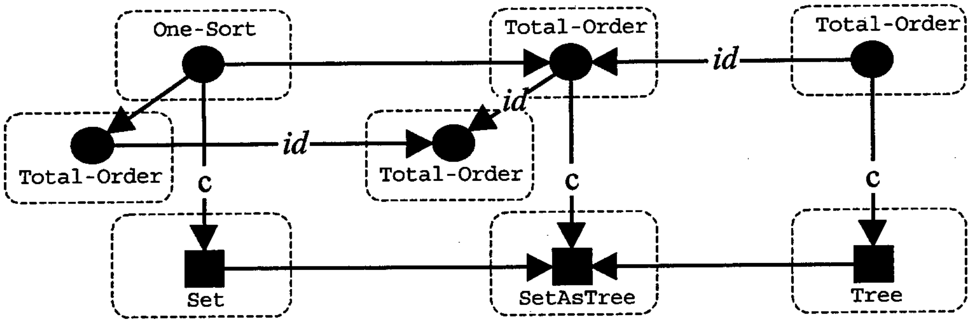
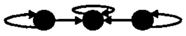
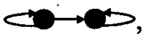


Figure 6-8. Set-TotalOrder-as-Tree constrained parameterized diagram interpretation

6.2 *Applying Design Information*

In diagram refinement, in order to apply the design information (*Spec* interpretations) the domain (source) specification of the *Spec* interpretation had to match exactly a specification within the requirement diagram being refined. In contrast, the domain (source) diagram of a diagram interpretation need not exactly match some part of the requirement diagram; there must only exist a diagram morphism between the domain diagram and the requirement diagram. Application of the design information is then accomplished via a pushout in *dSpec*. The design information that is represented as a diagram interpretation can be applied to (the structure of) requirement specifications as well as to other diagram interpretations.

6.2.1 Repartitioning the design information

Before the design information contained in a diagram interpretation can be applied to a requirement diagram, it must be repartitioned so that the mediator and codomain diagrams are in the same partition. Every diagram interpretation is a partition (Definition 3.20) of the following shape  of the underlying flattened (Definition 3.18) *Spec* diagram. As such, every diagram interpretation can also be partitioned to have the following shape , where the definitional extension morphism (and its dom and cod diagram objects) have been partitioned to be in a single diagram. Thus the diagram interpretations in Figure 4-14, Figure 6-3, Figure 6-5, and Figure 6-7 can be partitioned into the *dSpec* diagrams in Figure 6-9.

The diagram on the left in each of the repartitioned diagram interpretations in Figure 6-9 represents the *preconditions* for applying the design information. These preconditions must be present in the requirement specification before the design information can be applied. The diagram on the right in each case represents the design information that is to be applied.

6.2.2 Using a *dSpec* pushout to apply design information

In order to apply the design information, one first forms a diagram morphism from the required preconditions diagram to the requirement diagram. This establishes that the preconditions for using the design information exist in the requirement diagram. One then takes the pushout of the resultant *dSpec* diagram, *RequirementDiagram←Precondition→DesignInfo*, as depicted in Figure 6-10. In that figure, a diagram morphism is formed between the preconditions for the design information and the requirement specification version *n*. Having established the relationship, the *dSpec* pushout (Definition 3.27) of the resulting *dSpec* diagram is taken, which merges the design information and the requirement specification *n* to form a new requirement specification version *n*+1.

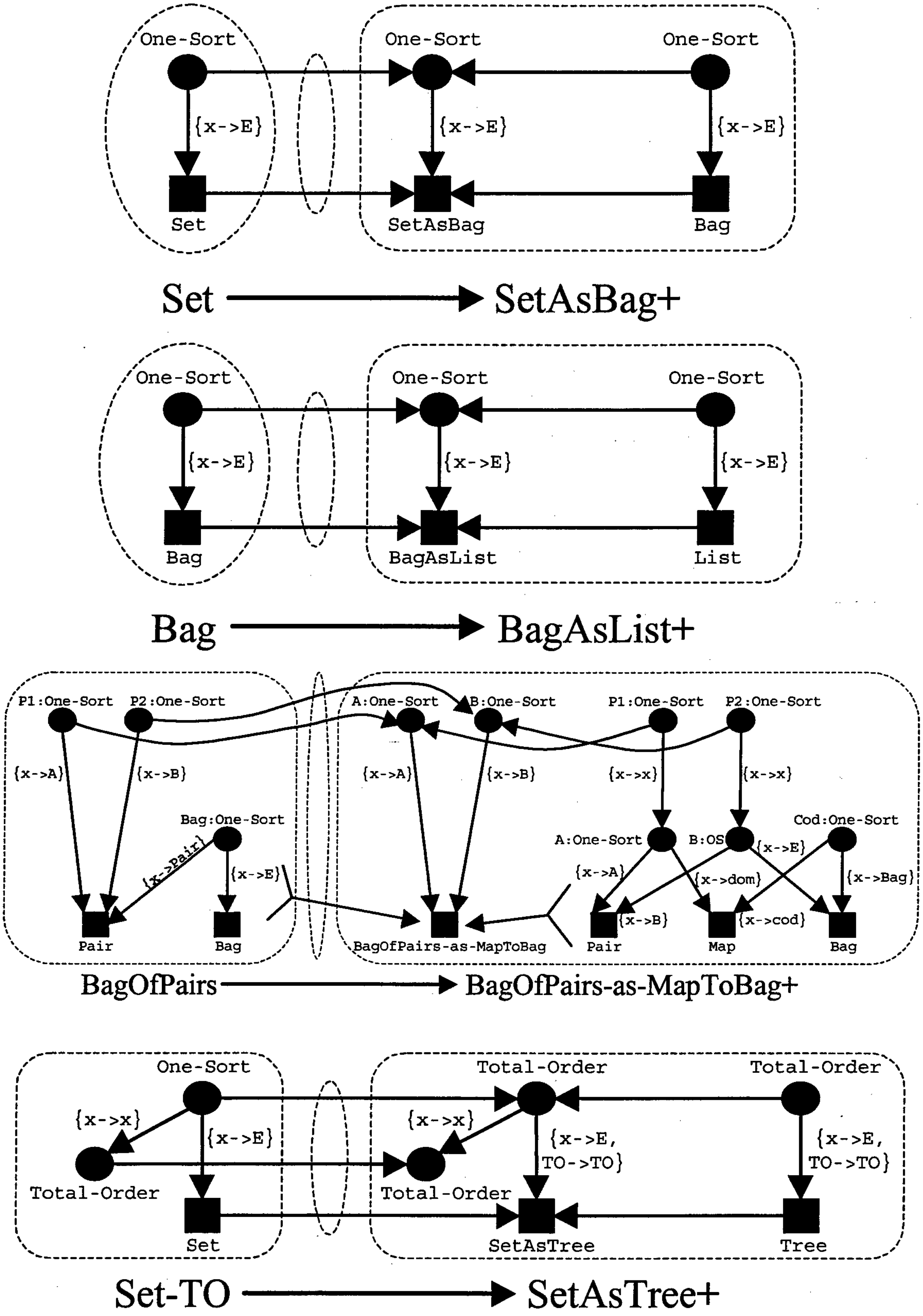


Figure 6-9. Repartitioned diagram interpretations as design information

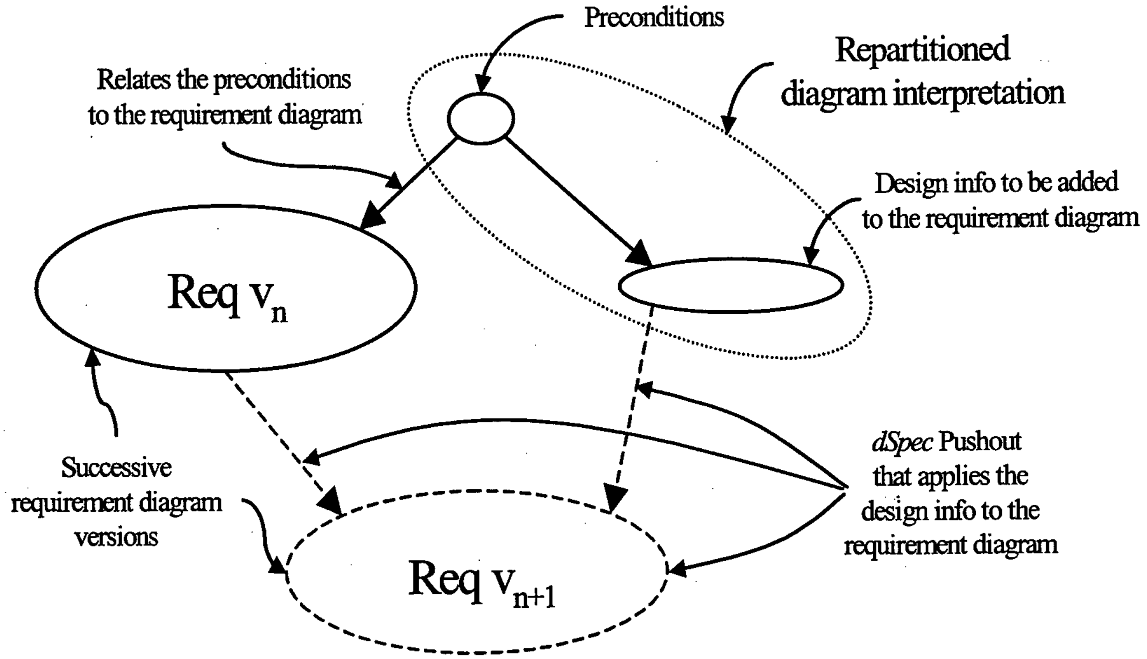

Preconditions
Relates the preconditions to the requirement diagram
Repartitioned diagram interpretation
Design info to be added to the requirement diagram
Req v_n
Successive requirement diagram versions
dSpec Pushout that applies the design info to the requirement diagram
Req v_{n+1}

Figure 6-10. Applying design information to a requirement specification

As an example application of diagram information, in Figure 6-11 the Set-TO to Tree interpretation has been applied to the simple requirement specification Set of Nat. Note that the repartitioned diagram interpretation Set-TO to Tree has been internally "flipped" from how it was represented in Figure 6-9 in order to have the arrows making up the diagram morphisms line up better. Also, some of the *Spec* arrows making up the diagram morphisms are labeled with *id* for the identity morphism so that the specifications in the resulting colimit object can more easily be related to the specifications in the diagram objects from which the colimit object was formed. The pushout object diagram that is the result of applying design information is "version 2" of the requirement specification. The requirement specification is still a Set-of-Nat but the design choice that the Set specification is to be implemented by a Tree specification has been integrated into the next version of the requirement specification. The application of a design choice results in a more detailed requirement specification as the new design information is now part of the requirements as well. In this case, the Tree –d→ SetAsTree definitional extension morphism indicates how a Tree ADT can be extended to implement a Set ADT. Thus one no longer has to worry about implementing a Set, one only has to implement a Tree. Any further refinements would be applied to the Tree specification, which has the Total-Order specification as its formal parameter.

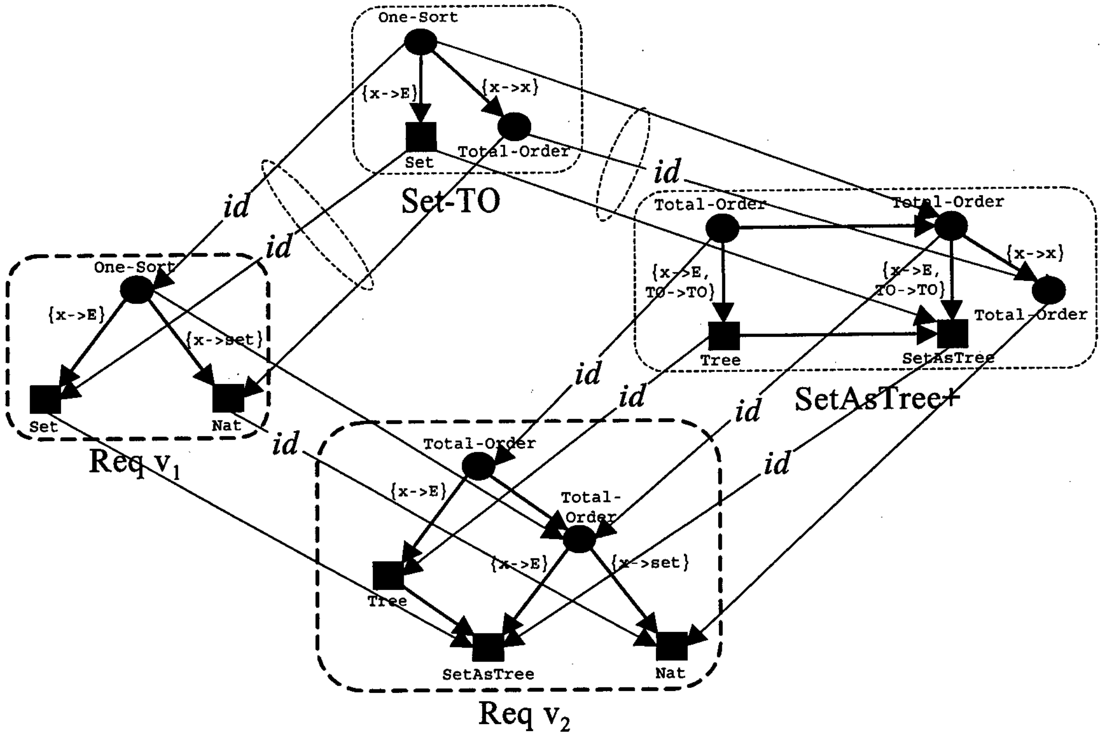


Figure 6-11. Example application of design information

Note the diagram morphism between the two versions of the requirement diagrams in Figure 6-11 (and the abstraction in Figure 6-10). Each successive design decision is applied to the results of the previous design decision, as the application of design information literally produces a new requirement diagram. To paraphrase [SB82], requirements and design are inevitably intertwined. With each application of design information, the specifications in the requirements diagram become more detailed and more like design specifications. The sequence of diagram morphisms between the versions of the requirement diagrams form a chain enabling the original requirements to be related directly to the final design.

Figure 6-12 depicts a sequence of design decisions being applied consecutively to a requirement diagram, which starts out as a Set of Set of Flags. The first design choice is to represent the outer Set as a Bag. The second design choice is to represent the Bag as a List. (The "Bag" requirement in Req v_2 is "new" based on the last design decision.) The third design decision is to represent the inner Set as a List. Rather than using the individual design decisions of Set ⇒Bag and Bag ⇒ List, a direct design decision Set ⇒ List is being applied. Section 6.2.3 addresses how design information composes so that Set ⇒ List can be formed from the existing design decisions Set ⇒ Bag and Bag ⇒ List. In the fourth design decision, the specification Flag has been refined to specification Nat.

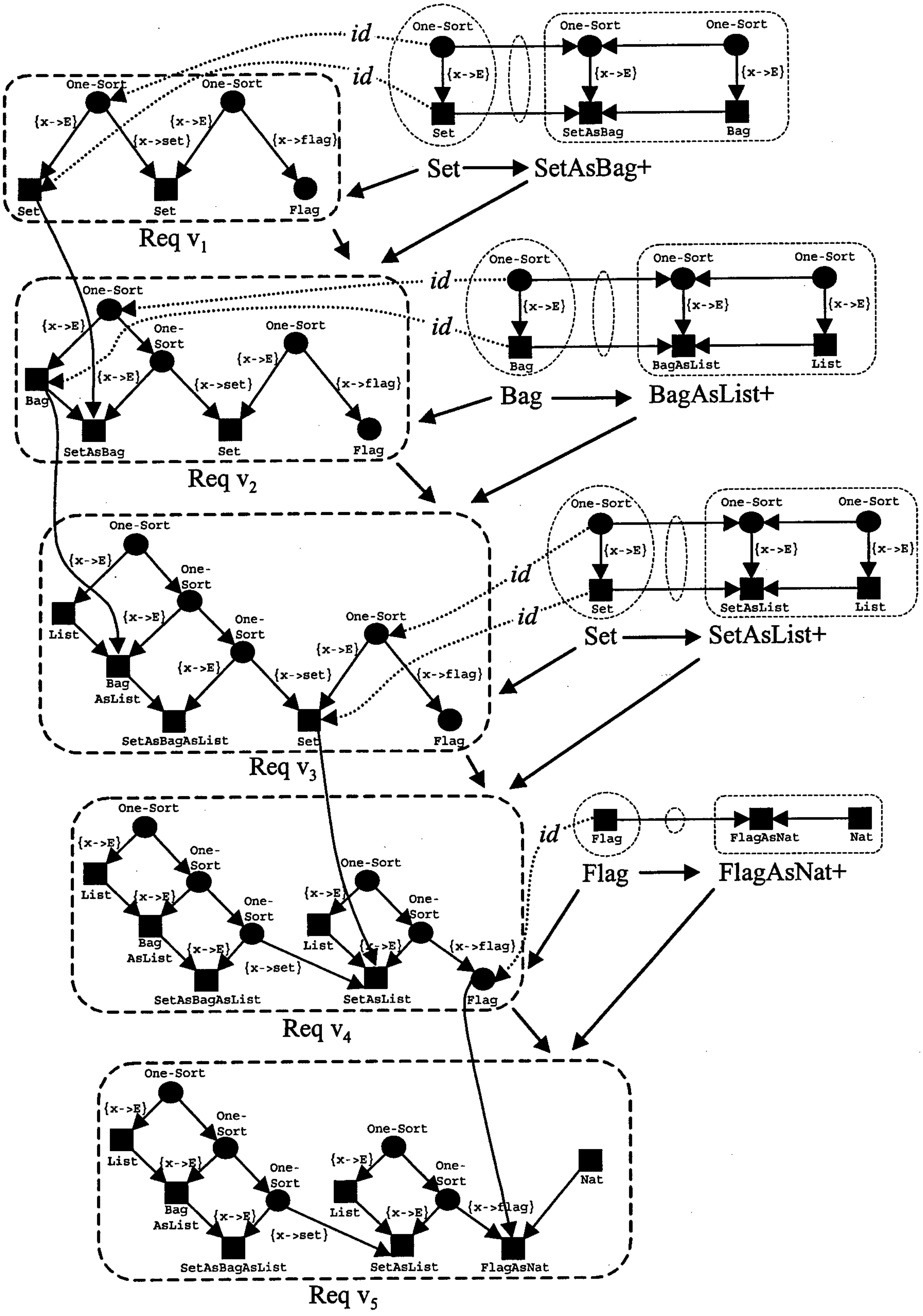


Figure 6-12. A sequence of design decisions leads to a sequence of requirement diagrams

Note that in the diagram morphism Req v_2→Req v_3 in Figure 6-12 that the SetAsBag specification has been changed to the SetAsBagAsList specification. This is because the addition of the Bag⇒List design information "rippled" through to the SetAsBag specification, which is what was wanted. When a specification in a diagram is changed by adding additional design information to it, any specifications "downstream" of it will also be affected.

In some cases design information must be applied in a specific order. In this case it did not matter whether the inner Set or outer Set was refined first. However, the choice to represent the outer Set as a Bag had to proceed the choice to represent the Bag as a List. This demonstrates how earlier design decisions can influence and make available additional design decisions.

An alternate design decision in Figure 6-12 could have been to represent the inner Set as a Tree using the Set-TO to Tree design choice. Initially this design choice is not possible as there is no total order operation defined in the Flag specification. However, once the design decision is made to implement the Flag specification by the Nat specification, the Flag sort will have a total order operation defined over it that is derived from its implementation as a Nat. Thus if the Flag as Nat design decision is made earlier, this opens up the possibility of storing the set of Flags as a sorted Tree that makes it quicker to access the elements. (Although for this simple case the Flag sort only has three values making this a "bad" design choice; a bit vector may be better).

Note that the application of the design choice Flag⇒Nat is easily accomplished without causing the problems experienced by the diagram refinement approach.

6.2.3 Applying design information to design information

Rather than apply all design information to a requirement specification, it ought to be possible to apply design information to itself. This capability enables larger "chunks" of design information to be applied as a unit. As the accompanying text for Figure 6-12 implied, rather than having to apply a Set-to-Bag refinement and then a Bag-to-List refinement, it should be possible to join those two design decisions into a single one so that a Set-to-List refinement can be applied. While Definition 4.8 ensures that diagram interpretations compose, it is not necessarily the case that the properties associated with the different classifications of diagram interpretations compose. This section develops the underlying theory of composing design information and provides several examples of composing design information.

Simple diagram interpretation composition

The class of simple design information composes using normal diagram interpretation composition (Definition 4.8). Given that A⇒B and B⇒C are instances of simple design information, then A⇒C is also, as "variability" cannot be introduced during the composition.

Parameterized diagram interpretation composition

The class of parameterized design information composes as depicted in Figure 6-13. At the top of that figure are two parameterized diagram interpretations with a common domain/codomain. In the middle, two separate pushouts are taken to form the formal parameter and body of the new mediator of the design information. The arrow between the new formal parameter and the body is the universal arrow between a colimit object and any other cocone object of a given diagram. At the bottom of Figure 6-13, arrow composition is used to connect the new mediator with the domain and codomain diagrams creating a new parameterized diagram interpretation.

The arrow that connects the New Mediator formal parameter diagram to the New Mediator body diagram in Figure 6-13 will always be a conservative extension given that it is two parameterized diagram interpretations that are being composed. Notice that the commuting square connecting the right Mediator diagram to the New Mediator diagram will necessarily look like the commuting square on the left in Figure 6-14. (Where the property of the diagram morphism from the formal parameter to the body is not known, but the other diagram morphism properties must be as depicted.) In the middle diagram in Figure 6-14 a pushout object has been formed and the arrows to it will be definitional and conservative extension morphisms, as those properties reflect across pushout squares (Proposition 4.3). There must also exist a universal arrow as indicated whose properties are also not yet known. On the right in Figure 6-14 the universal arrow has been determined to be a definitional extension morphism, as the two definitional extension morphisms compose to form a definitional extension morphism. Finally, the unknown New Mediator diagram morphism is determined to be a conservative extension morphism based on it being equivalent to the composition of a conservative extension and definitional extension morphism. Thus normal diagram composition is all that is needed to compose two parameterized diagram interpretations.

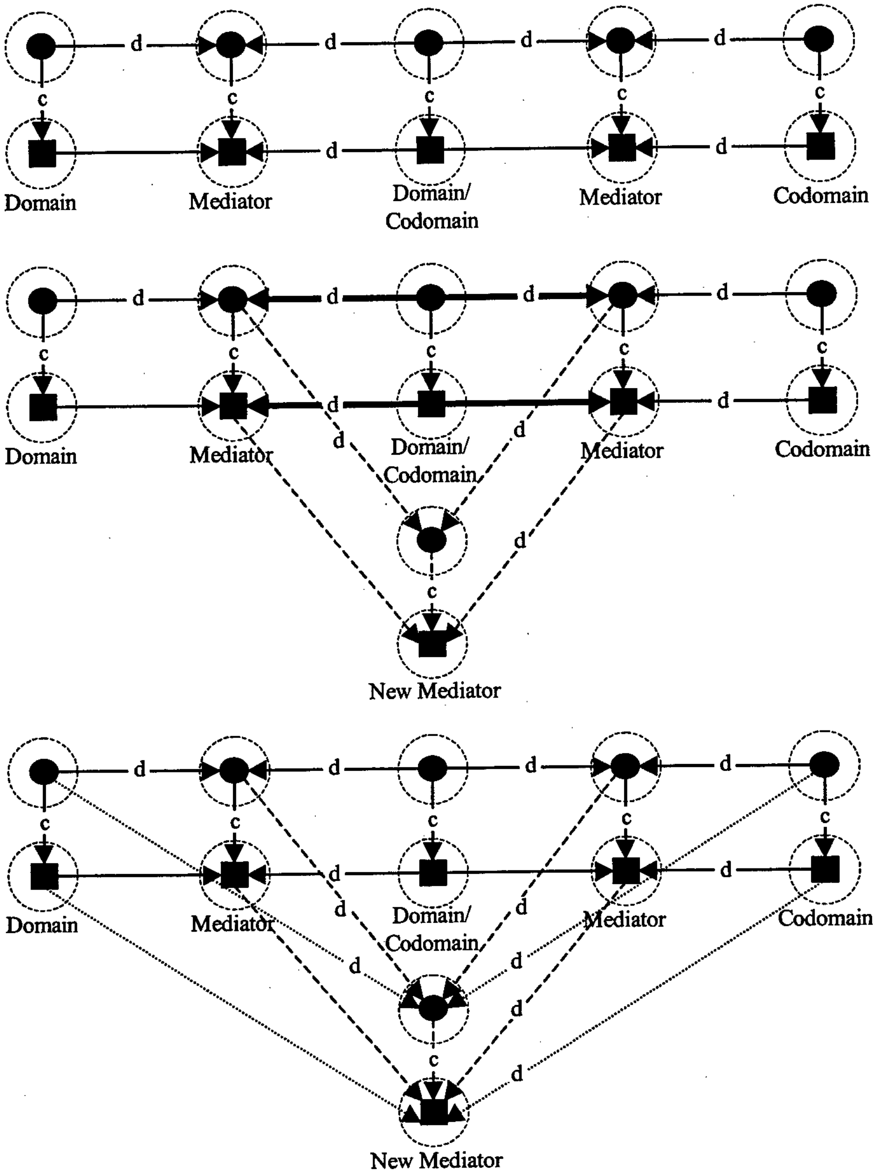


Figure 6-13. Composing parameterized dInterpretations

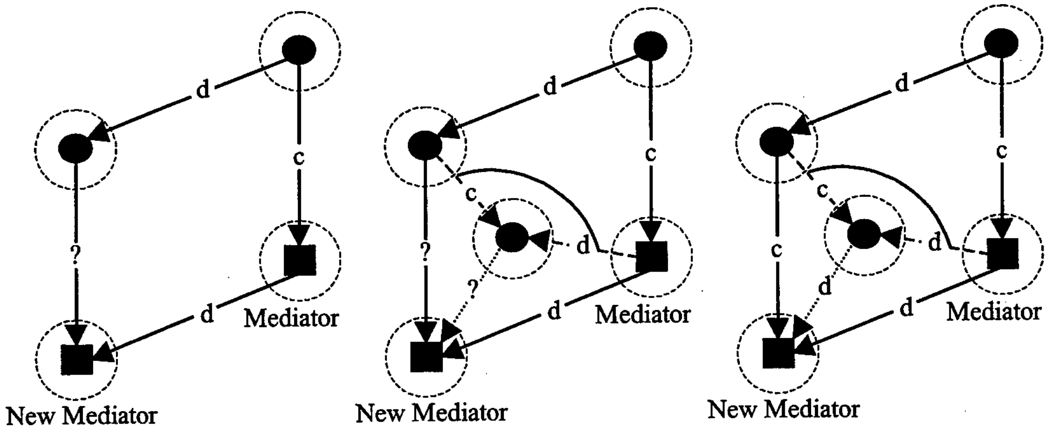


Figure 6-14. Formal parameter → Body arrow of the New Mediator is a conservative extension

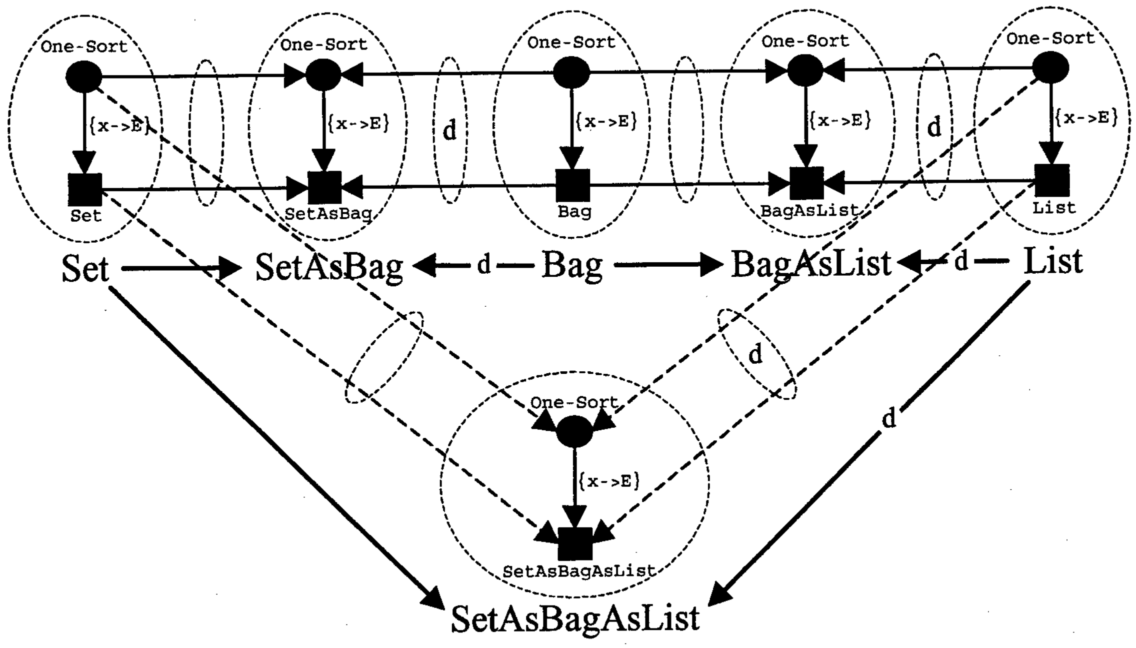


Figure 6-15. Example composition of parameterized dInterpretations

An example composition of parameterized diagram interpretations is depicted in Figure 6-15. That figure depicts the results of composing the Set⇒Bag and Bag⇒List interpretations.

Constrained parameterized diagram interpretation composition

Both simple and parameterized diagram interpretations compose with the normal diagram interpretation mechanism from Definition 4.8. This approach only partially works with constrained parameterized diagram interpretations. When a constrained parameterized diagram Interpretation is being applied to a requirement specification (diagram), the constraint and the domain parameterized diagram are treated as a single unit (as are the mediator and codomain parameterized diagrams and the mediator constraint), see Figure 6-11. This is possible because in order for the design information to be applied correctly the properties in the constraint must be present in the requirement specification. That is, the parameterized diagram must have been instantiated in the requirement specification and therefore it is possible to determine whether or not the given requirement instantiation satisfies the properties in the constraint. However, when composing constrained parameterized diagram interpretations no instantiation has taken place and therefore the constraining diagrams must be treated separately from the domain, mediator and codomain parameterized diagrams. Constrained parameterized diagrams can also be composed on the left (Figure 6-16) or on the right (Figure 6-17) with "normal" parameterized diagrams.

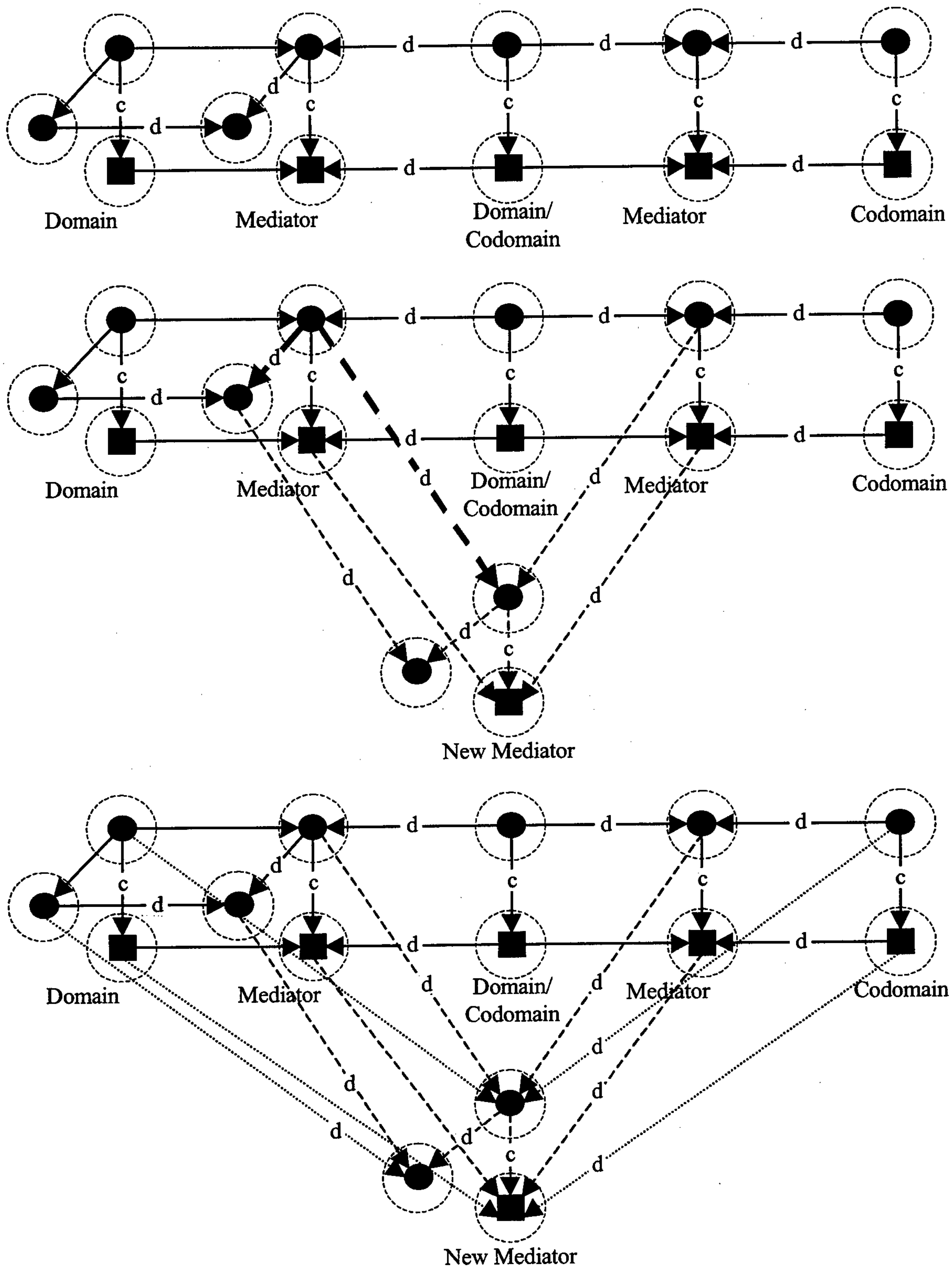


Figure 6-16. Composition with a (left) constrained parameterized dInterpretation

When composing a constrained parameterized diagram interpretation on the left with a parameterized diagram interpretation, as depicted in the top diagram in Figure 6-16, the constraint is easily propagated to the composed diagram interpretation. The middle diagram in Figure 6-16 depicts how a pushout factors the constraint through to the new mediator. The bottom diagram in Figure 6-16 depicts how morphism composition completes the new composed constrained parameterized diagram interpretation.

In Figure 6-16, where the constraining diagram is on the left parameterized diagram interpretation, the constraint is propagated via a simple pushout. When composing a constrained parameterized diagram interpretation on the right with a parameterized diagram interpretation, as depicted in the top diagram in Figure 6-17, the constraint can also be propagated. Note that in the top diagram in Figure 6-17 the two (domain/codomain) parameterized diagrams were joined based on their common formal parameters and bodies and that the constraint placed on the right domain parameterized diagram is now constraining the combined domain/codomain parameterized diagram. The constraint will not occur in the left parameterized diagram interpretation, yet in order to compose the two it must be propagated to the domain parameterized diagram of the left parameterized diagram interpretation.

The top diagram in Figure 6-17 also depicts how a pushout and a composition of morphisms factors the constraint from the domain/codomain parameterized diagram through to the domain of the first diagram interpretation. In the second diagram in Figure 6-17 a pushout is used to factor the constraint through to the new mediator. In the third diagram in Figure 6-17 an additional pushout is used to extend the new mediator constraint (if needed) and to ensure that it has a definitional extension morphism to it from the new domain constraint that was constructed in the top diagram. The bottom diagram in Figure 6-17 depicts how morphism composition completes the newly composed constrained parameterized diagram interpretation. Thus while the formal parameters and bodies of the composed diagram interpretations are always computed via the normal diagram interpretation composition, any constraining diagrams must be handled separately as depicted in Figure 6-16 and Figure 6-17.

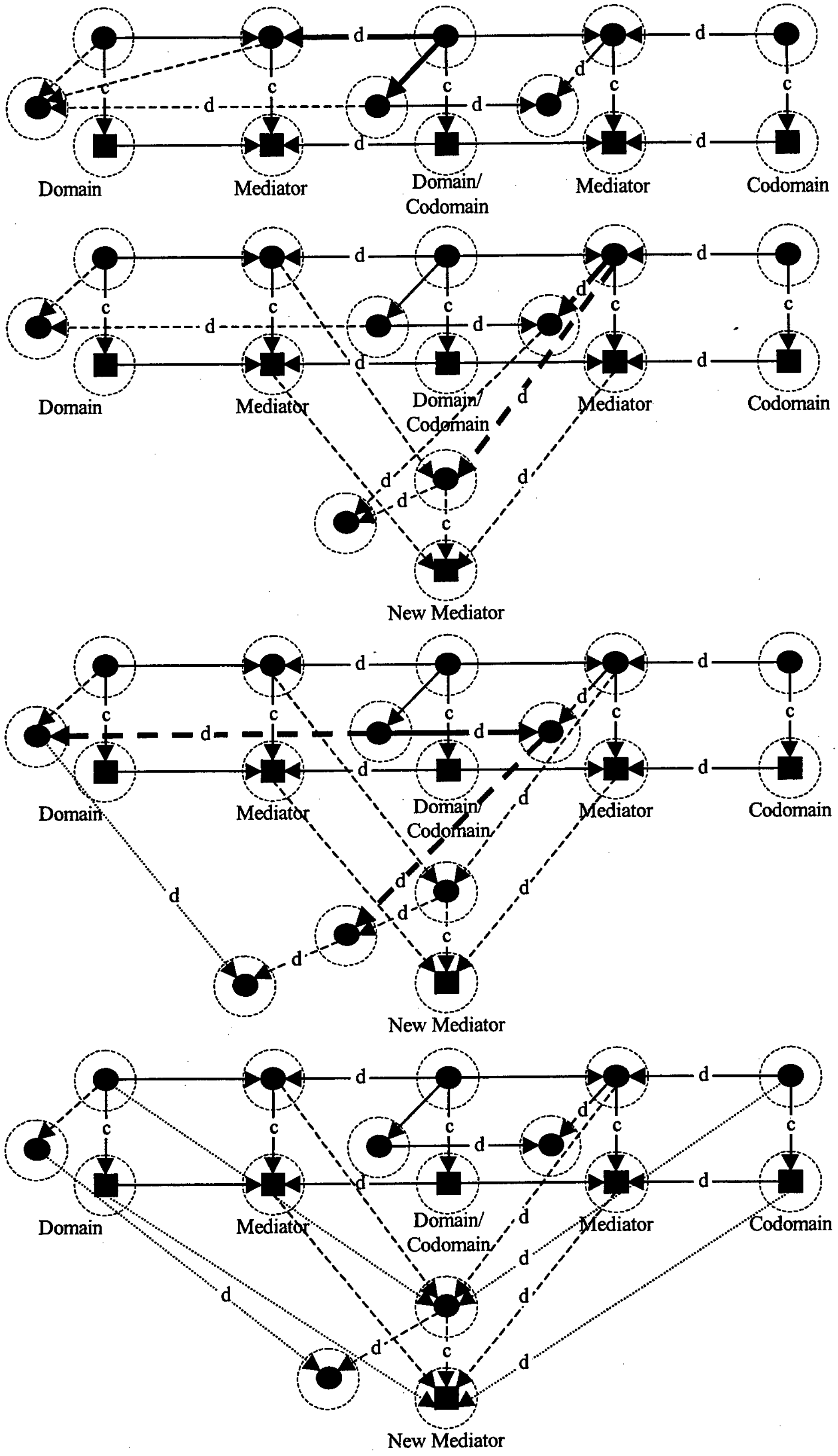


Figure 6-17. Composition with a (right) constrained parameterized dInterpretation

Composing two constrained parameterized diagram interpretations is also possible. In Figure 6-18, a pushout is used to factor the right constraint "over" the left constraint. Then the morphism from the left diagram interpretation's domain formal parameter to the new constraint is constructed via the composition of three morphisms. The new constraint on the left domain formal parameter in Figure 6-18 necessarily contains the old constraint on the left domain formal parameter as well as the constraint on the right domain.

Compare the diagram in Figure 6-18 with the diagram at the top of Figure 6-17 to see how the right constraint was placed over the left constraint. The rest of the sequence in Figure 6-17 can now be used to complete the construction of the constrained parameterized diagram interpretation in Figure 6-18.

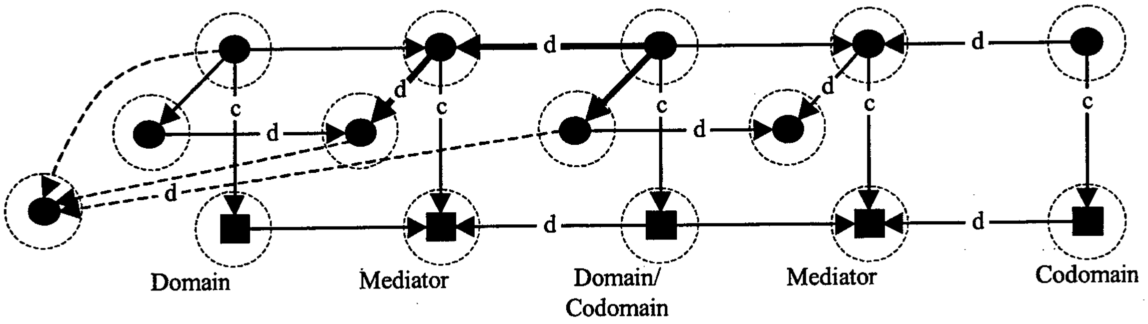


Figure 6-18. Composition with two constrained parameterized dInterpretations

An example of parameterized diagram interpretation composition that involves constraints is depicted in Figure 6-19. Normal diagram interpretation composition is used to construct the new mediator. The constraint is propagated to the new parameterized diagram interpretation via a succession of colimits that involve the formal parameter diagrams as described earlier.

In summary, all three classes of design information compose by normal diagram interpretation with the exception of the constraint information. That information can be composed by operating strictly over the constraint and formal parameter sub-diagrams.

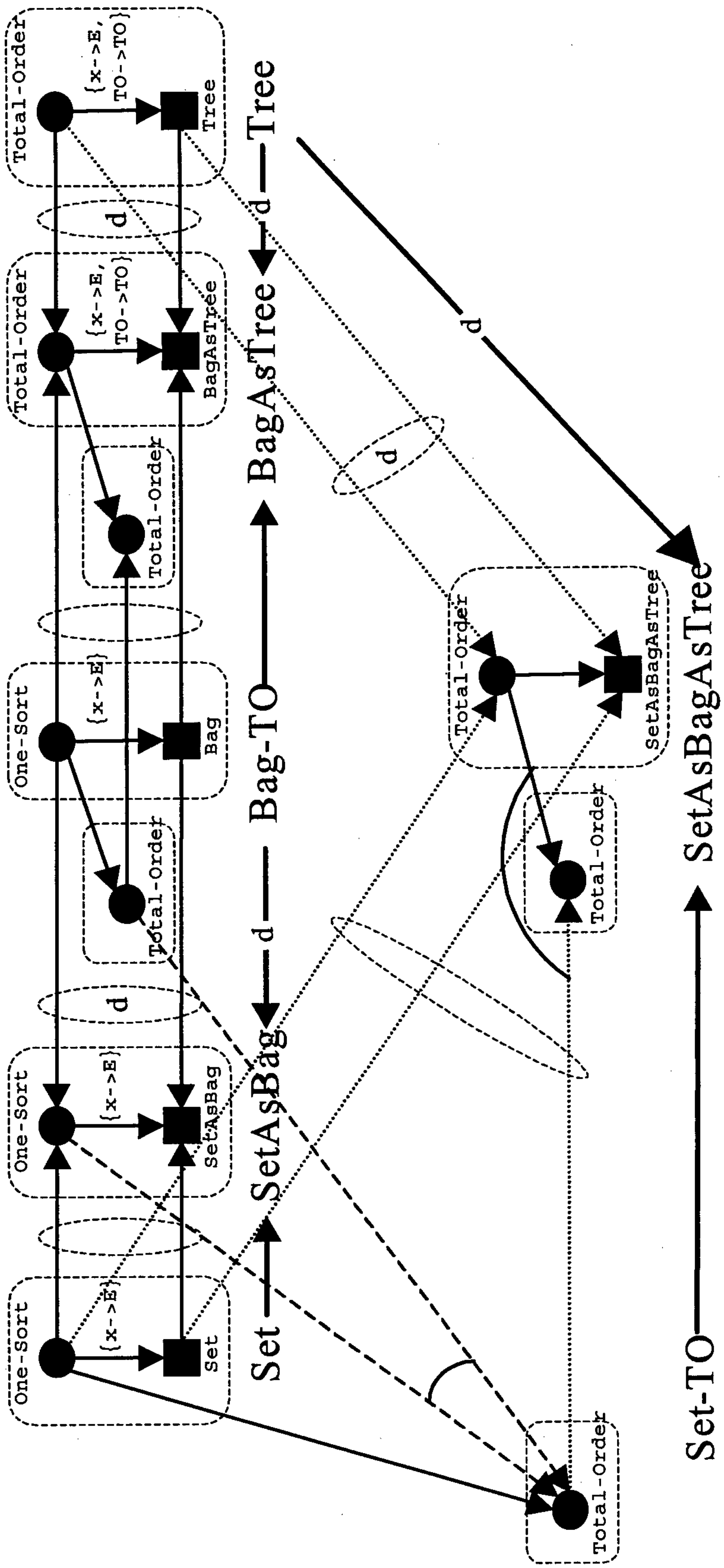


Figure 6-19. Example composition of parameterized dInterpretations with (right) constraint

6.3 Simplifying the Structure of Design Information

The two previous sections have described the theory of representing, composing and applying design information using diagram interpretations (and repartitioned diagram interpretations) and presented several examples. As can be seen, the number of objects and arrows in the diagrams for even these small examples can be a bit daunting. There are a few short cuts that one can take in representing, composing and applying design information that make it more palatable than has been previously depicted.

Design information must only be proven correct once prior to being stored in a design library and made available for reuse, and it need not be developed and represented in diagram interpretation form. In certain contexts the design information can be represented in a simpler fashion than has been previously presented.

6.3.1 Motivating example

As a motivating example, Figure 6-20 depicts the effects of a pair of structure-reducing operations that start with the final requirement diagram from Figure 6-12. In the transition from Req v_5 to Req v_6, the BagAsList specification is removed because it is the intermediate specification between a pair of definitional extension morphisms. (The pair of d-morphisms List $\xrightarrow{d}$ BagAsList $\xrightarrow{d}$ SetAsBagAsList is reduced to the definitional extension morphism List $\xrightarrow{d}$ SetAsBagAsList, as the intermediate definitional extension does not really add any information to the requirement diagram. It has served its purpose so to speak.) This reduction induces an obvious diagram morphism from diagram Req v_5 to diagram Req v_6. As no new structure or design information has been added, it essentially is a no-op.

In the transition from Req v_6 to Req v_7 all nodes in the diagram with an identity morphism between them are folded together (Definition 3.23). Thus the multiple One-Sort specifications are removed. As the fold operation induces a diagram morphism between the diagrams, there is an obvious diagram morphism from diagram Req v_6 to diagram Req v_7. The requirement specification Req v_7 is a valid refinement of the requirement specification Req v_5. Req v_7 indicates that a Set of Set of Flags is to be implemented with the following smaller refinements: The outer Set is to be implemented as a List, the inner Set is to be implemented as a List, and Flags are to be implemented as a subsort of the natural numbers.

The final version of the requirement diagram in Figure 6-20 is easier to comprehend and work with than the requirement diagram prior to the structure reductions. However, instead of reducing the requirement specification structure after design information is applied, it may be better if the design information structure itself is reduced. Reducing the structure of the design information means that it can be developed and applied with less work. Even when the design information is reduced, however, cases will still arise that introduce extraneous structure in the refined requirement specification.

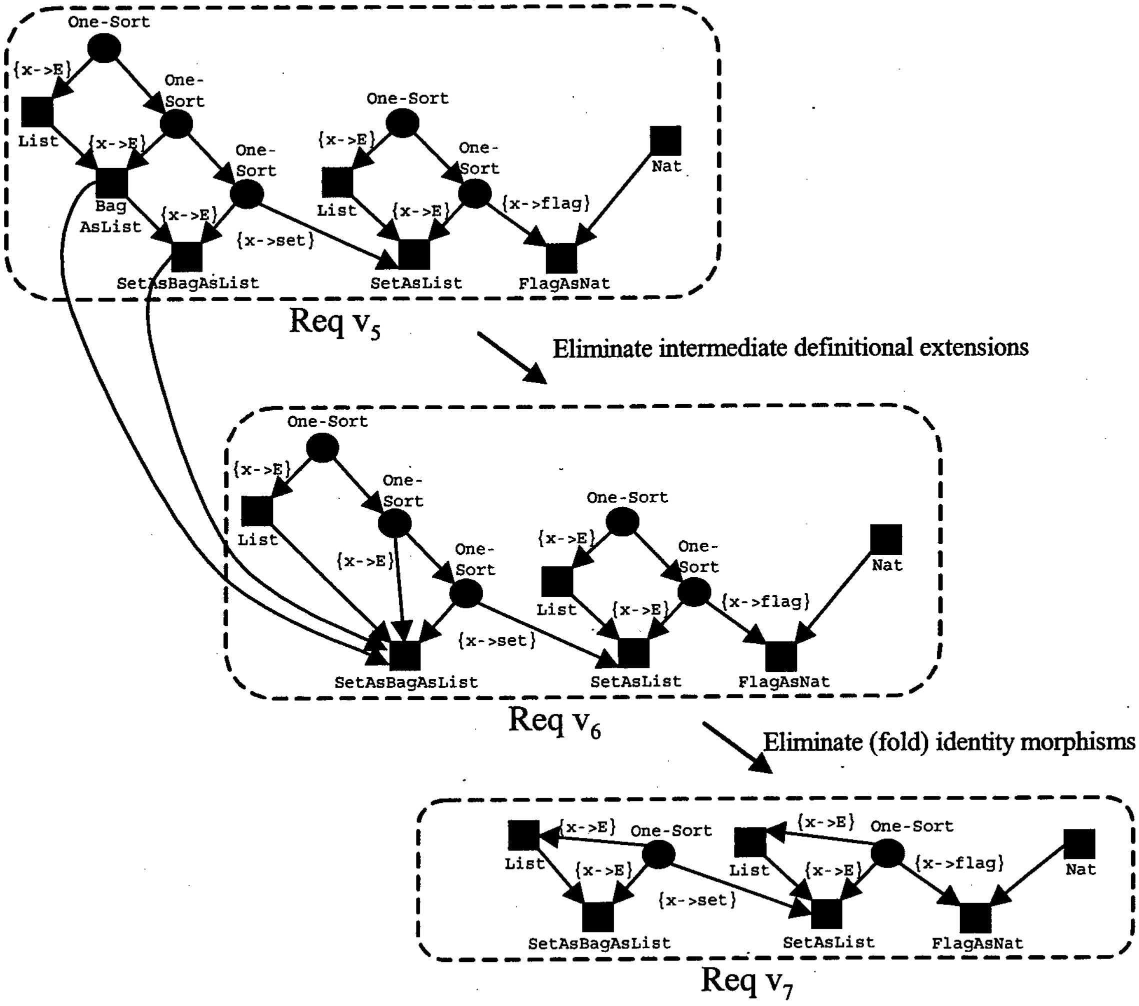


Figure 6-20. Reducing unneeded structure in the requirement diagram

6.3.2 Reducing design information structure

The classes of design information presented in Section 6.1.1 indicate the necessary properties for a simple, parameterized, or constrained parameterized diagram to be refined safely so that the refinement can be represented as design information. Section 6.2 described how that design information can be applied and composed. Assuming that a diagram interpretation meets the requirements of one of the classes of design information in Section 6.1.1, it can then be reduced in structure so that its application and composition can be accomplished in an easier fashion.

As an example, in Figure 6-21 and Figure 6-22 the four repartitioned diagram interpretations from Figure 6-9 have been reduced in structure. Specifically, in each case the formal parameters (in the single repartitioned design information part of the diagram) have had identity morphisms between them folded together. In the case of the BagofPairs diagram interpretation, additional nodes with identity morphisms between them were combined in addition to the formal parameters, as they were introduced to insure that all of the formal criteria for a parameterized diagram refinement were present. Even with all of the reduction in structure there still exists a diagram morphism to it from the preconditions part of the design information.

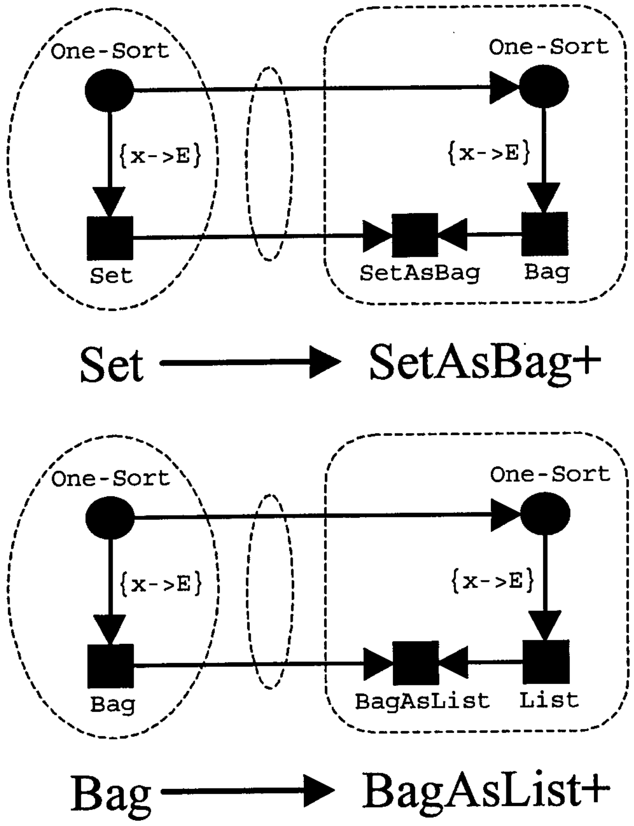


Figure 6-21. Reduced and repartitioned dInterpretations as design information I

Note, however, that in the Set to Tree diagram interpretation at the bottom of Figure 6-22, the identity morphism to the constraint (Total-Order) of the design information remains as a separate node. Maintaining the constraint separately is critical in terms of applying the design information, as without that separation too much of the structure of requirement specification will be combined when the design information is applied.

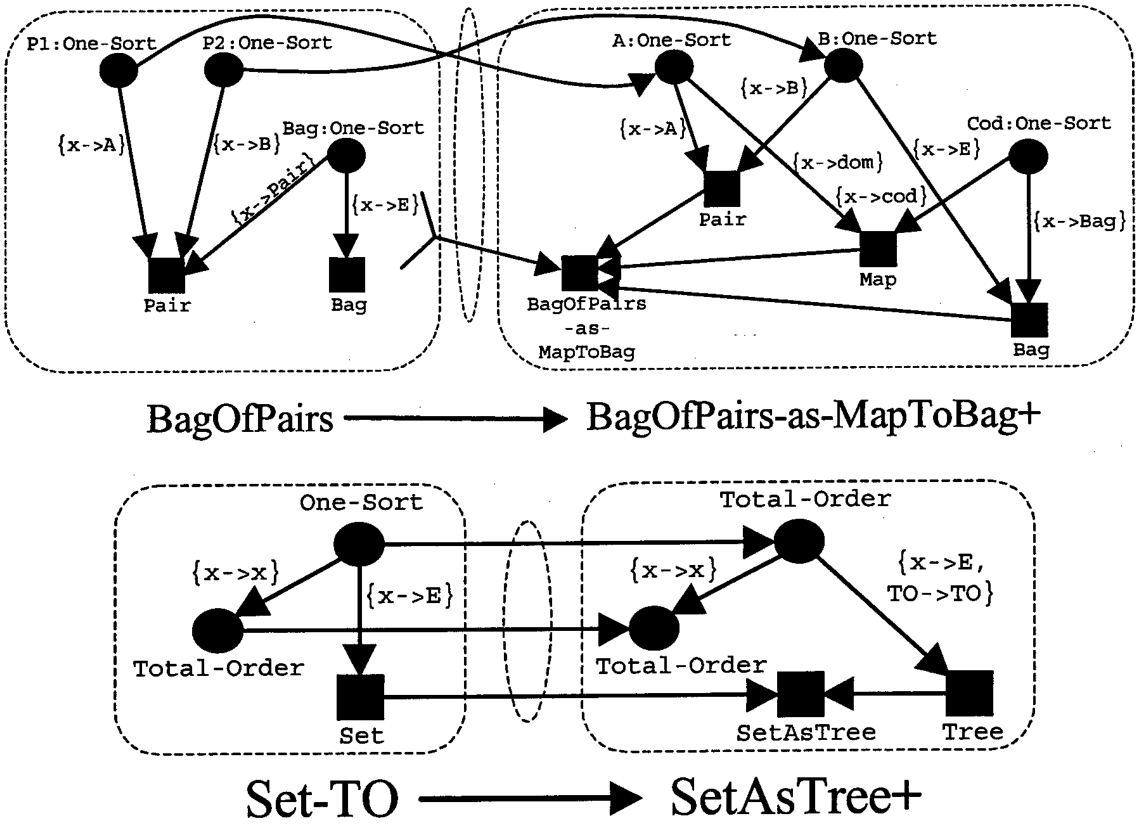


Figure 6-22. Reduced and repartitioned dInterpretations as design information II

An example of the application of reduced-structure design information is depicted in Figure 6-23; compare it to the non-reduced example depicted in Figure 6-11. Note how the pushout diagram, Req v_2, does not contain an extra One-Sort formal parameter specification as it did the previous version. The application of the reduced-structure design information avoided placing unneeded structure in the pushout requirement diagram.

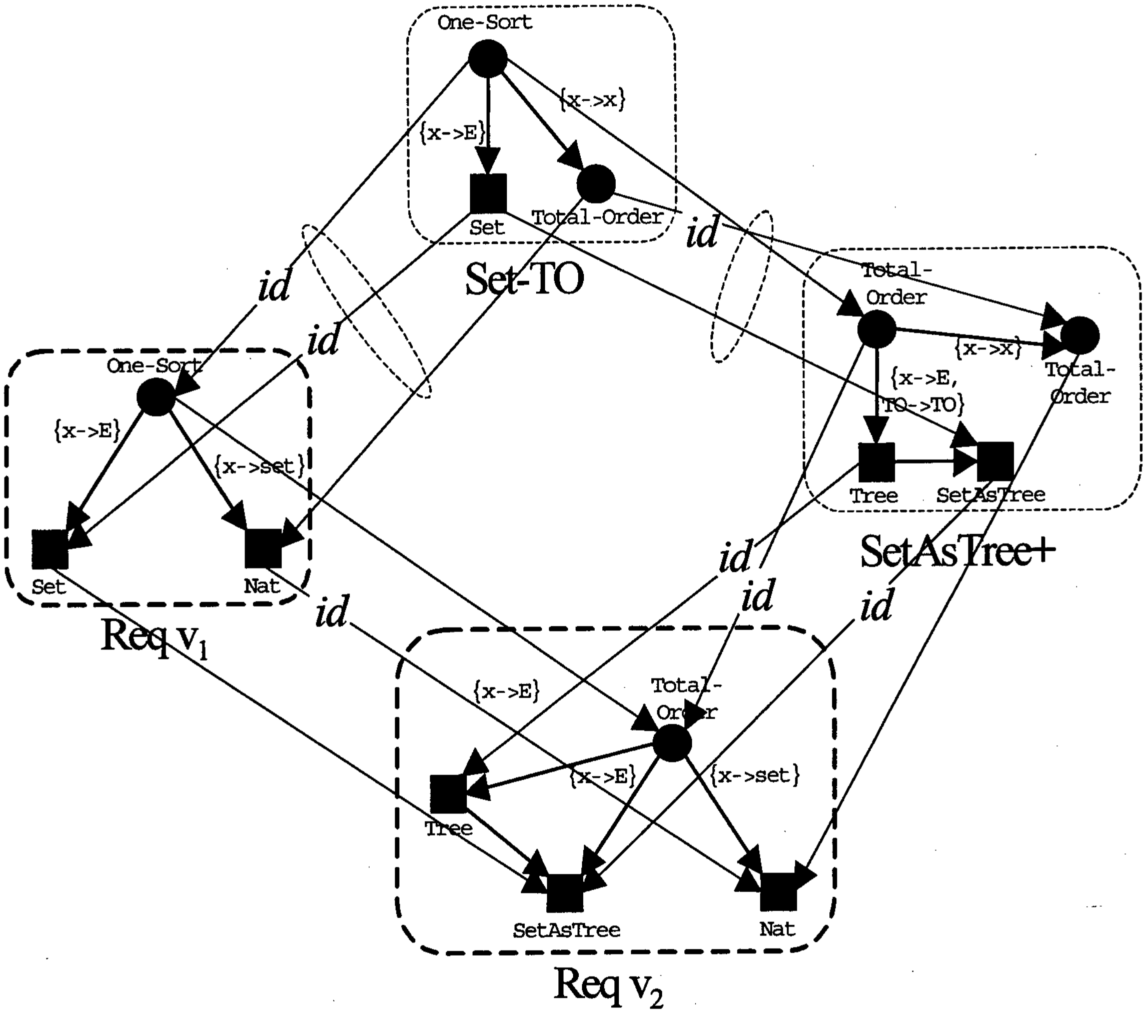


Figure 6-23. Example application of design information with reduced structure

Now compare Figure 6-23 with Figure 6-24, where the constraint specification (Total-Order) was combined with the formal parameter specification. Note how the instantiated specification Nat becomes the "formal parameter" in the pushout diagram object. Too much of the structure in the requirement specification has been collapsed. Thus when reducing the structure of design information it is acceptable to combine the formal parameters but the constraint must be maintained as a separate entity.

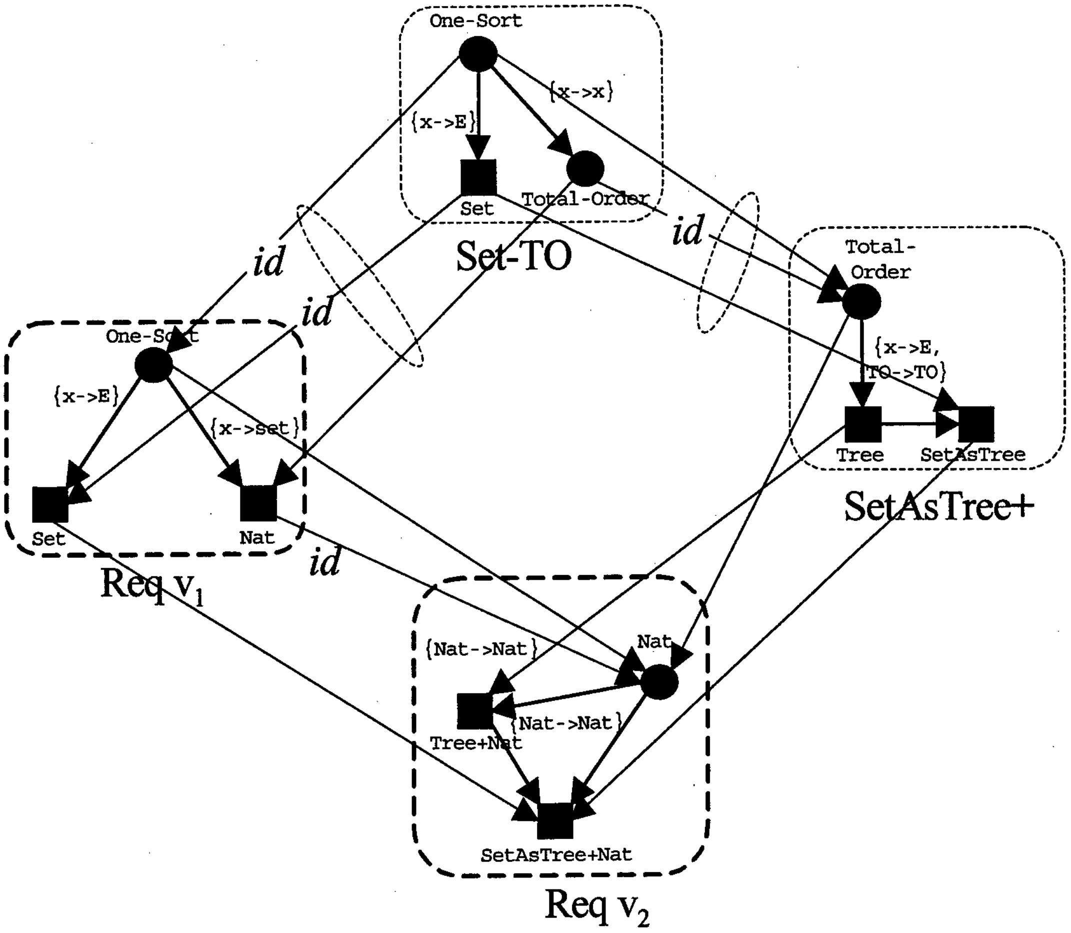


Figure 6-24. Example application of design information where structure is reduced too much

In some cases the composition of reduced-structure design information becomes simpler. In fact it becomes no different than applying design information to a requirement specification, as Figure 6-25 depicts. In Figure 6-25 a diagram morphism is formed between the "preconditions" of the BagAsList design choice to the design information in the SetAsBag design choice. Once the *dSpec* pushout of that diagram is taken, the resulting diagram, SetAsBagAsList+, is connected with the preconditions of the SetAsBag design choice via morphism composition. As this resulting composition still has unneeded structure, it can be reduced still further as depicted in Figure 6-26 where the unneeded intermediate definitional extension has been eliminated. Thus reduced-structure design information can still be composed despite it no longer having the form of a diagram interpretation.

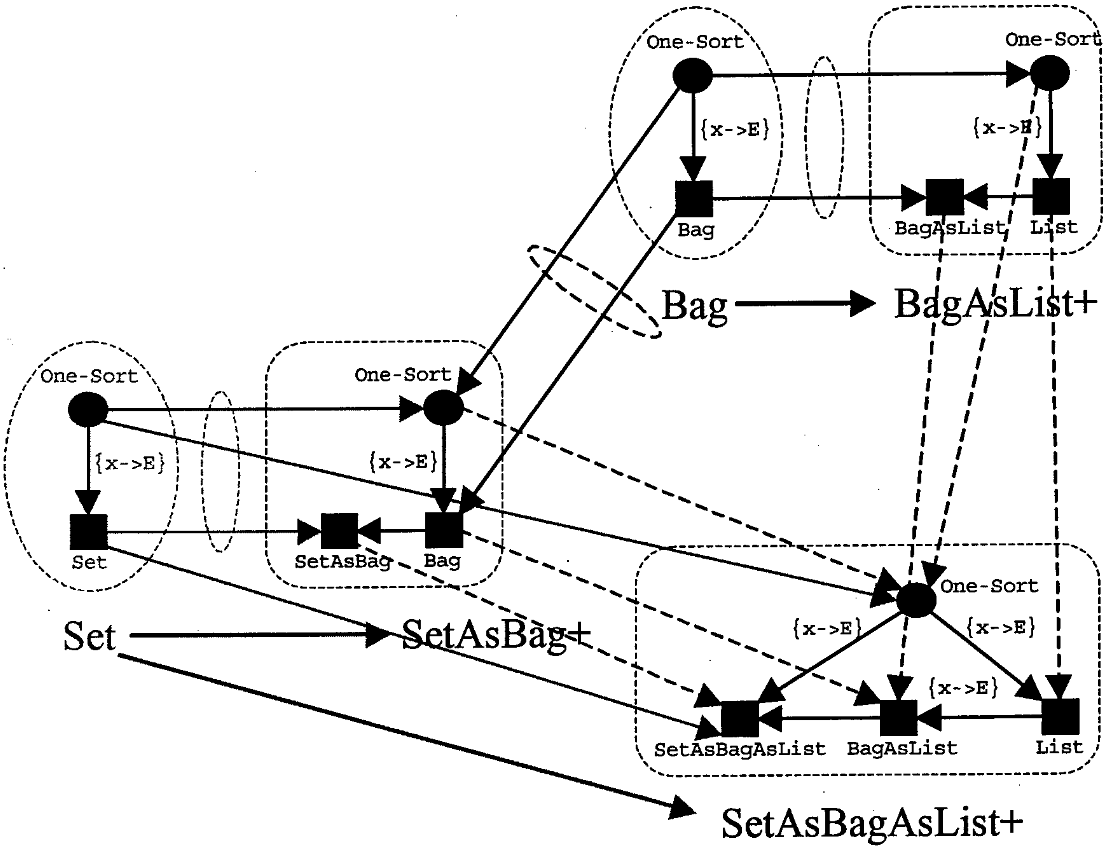


Figure 6-25. Composition of reduced-structure design information

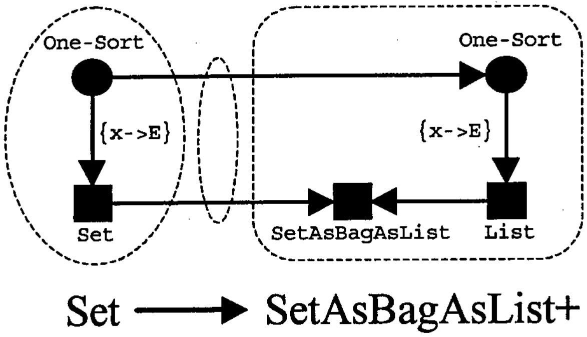


Figure 6-26. Unneeded structure in a requirement diagram is eliminated

As a final example of composing design information, Figure 6-27 depicts composing (reduced-structure) design information via pushout where the constraint is handled outside of the pushout mechanism. Once the structure of a constrained parameterized diagram interpretation has been reduced, constraints cannot be handled in as mechanical a fashion as described in Section 6.2.3. Composition of constrained parameterized diagram interpretations is still possible, though, as depicted in Figure 6-27.

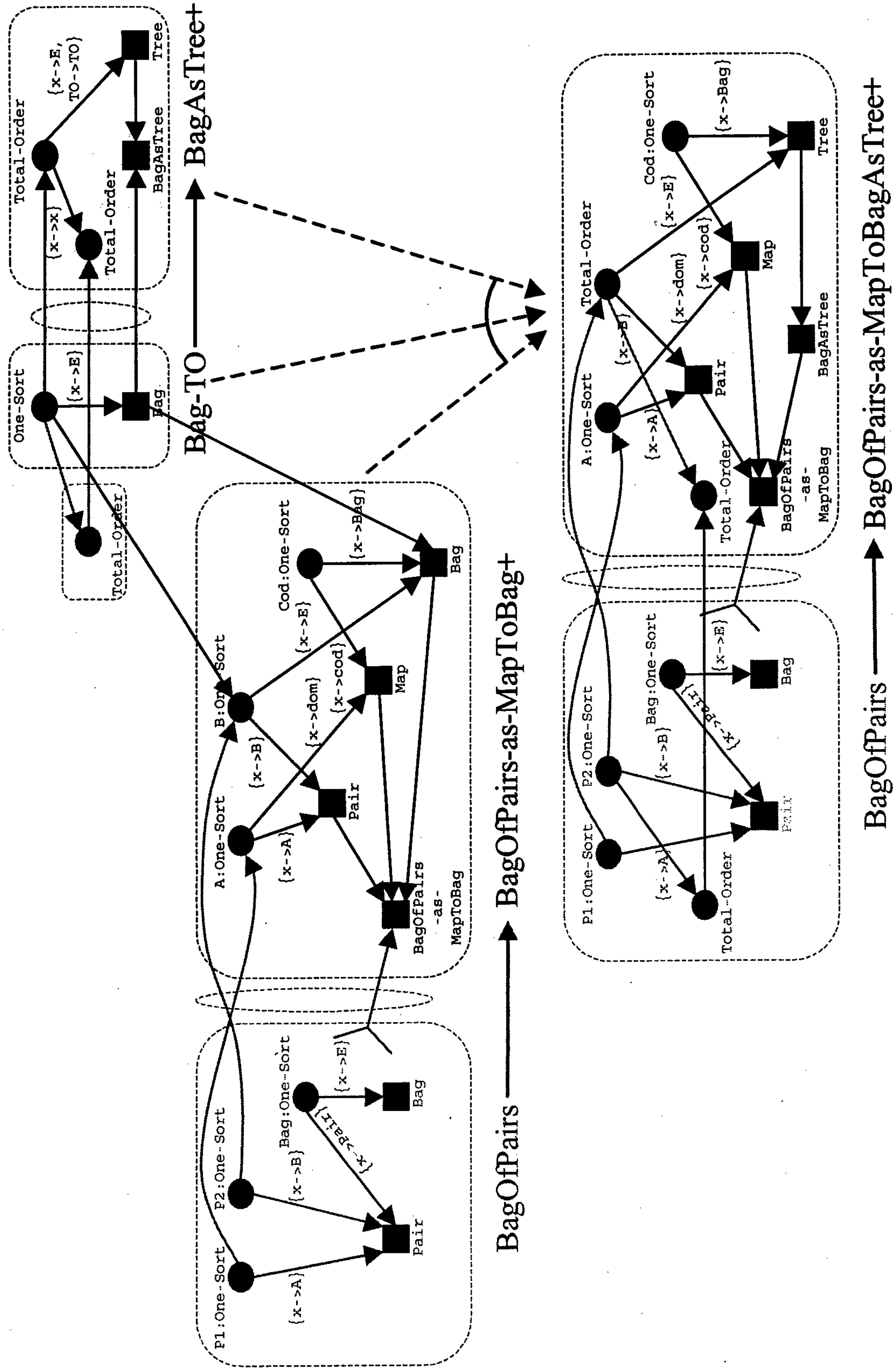


Figure 6-27. Complex composition of design information with constraints

6.4 Example Application of Design Information

This section refines a portion of that requirement diagram into a more concrete data representation. Specifically, the Bag of input arcs sub-diagram from the Petri Net requirement diagram is refined into a more concrete and implementable data structure in Figure 6-28 and Figure 6-29.

The first design choice in Figure 6-28 is to represent the Place and Transition sorts as natural numbers. As the Place and Transition specifications are isomorphic to the One-Sort specification, this addition of design information is fairly trivial. Note that the refinement of both Place and Transition is done in one large colimit, demonstrating that multiple, non-conflicting design choices can be accomplished at the same time.

The second design decision is to implement the Bag of input arcs (pairs of places and transitions) as a Map from a place to a Bag of transitions. Note that this design choice has appreciably changed the structure of the requirement diagram. Such a refinement is not possible when using the diagram refinement approach. The Bag specification in the diagram BagOfInputArcs_2 is completely unrelated to the Bag specification in BagOfInputArcs_3. The original Bag specification is only an emergent property (via a definitional extension) of the new data structure, which is a Map to a Bag, i.e. the BagOfPairs-as-MapToBag specification. The refinement continues in Figure 6-29.

The third design decision refines a Map to an Array as the constraint on the codomain parameter specification of the map (being a discrete linear order, Di-LO, Figure C-3) has been met by the PlaceAsNat actual parameter in the requirement diagram. In the new requirement diagram, the pushout diagram BagOfOnputArcs_4, note that the specification BagOfPairs-as-MapAsArrayToBag has also been effected as the design information MapAsArray has rippled through to it, which is what was wanted.

The fourth design decision refines a Bag to a Tree as the constraint on the formal parameter of Bag (being a total order) has been met by the TransitionAsNat actual parameter in the requirement diagram. Again, in the new requirement diagram the design information has rippled through to the BagOfPairs-as-MapAsArrayToBagAsTree specification; this ripple through of the design information is exactly what is wanted.

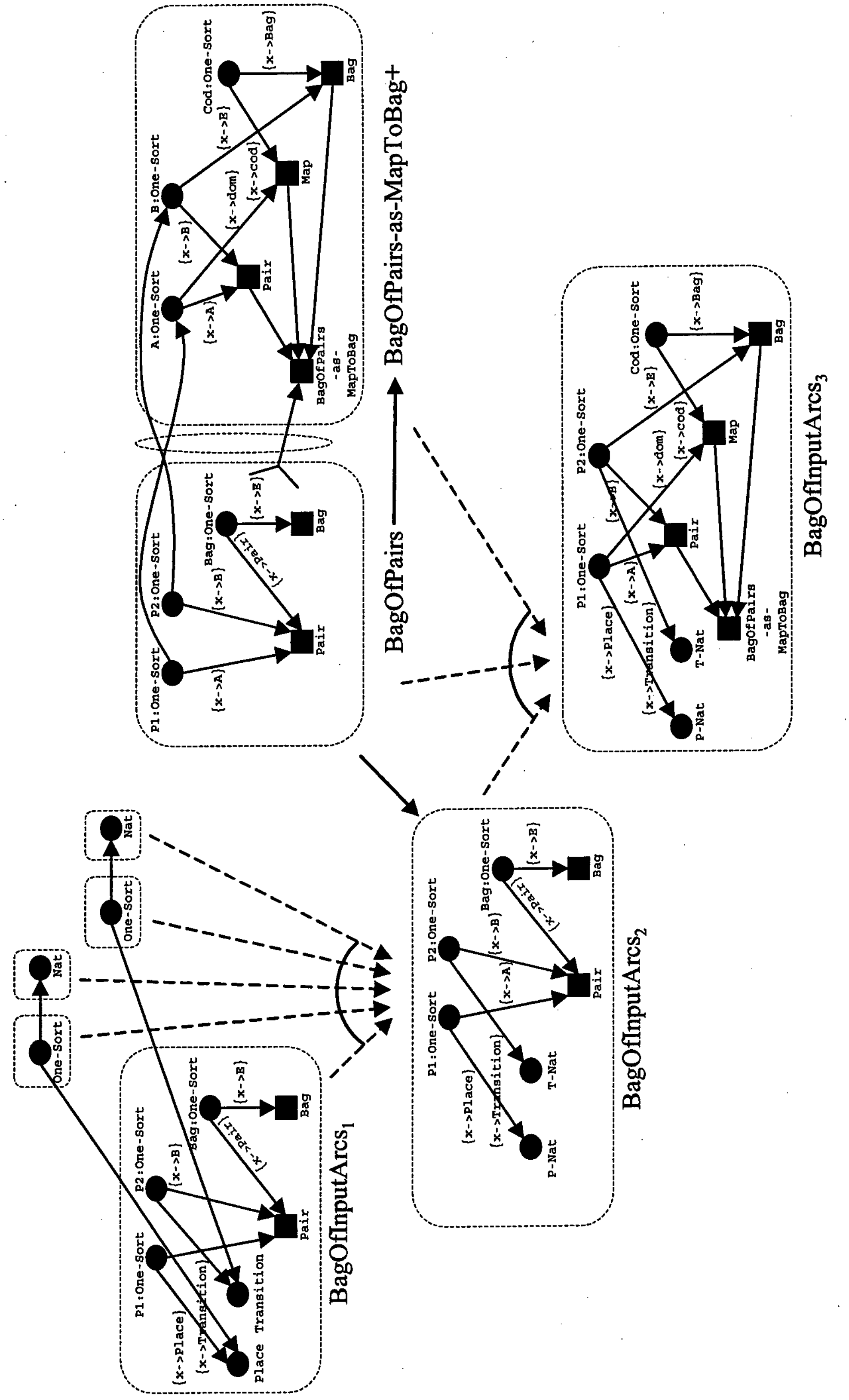


Figure 6-28. Petri Net data structure example I

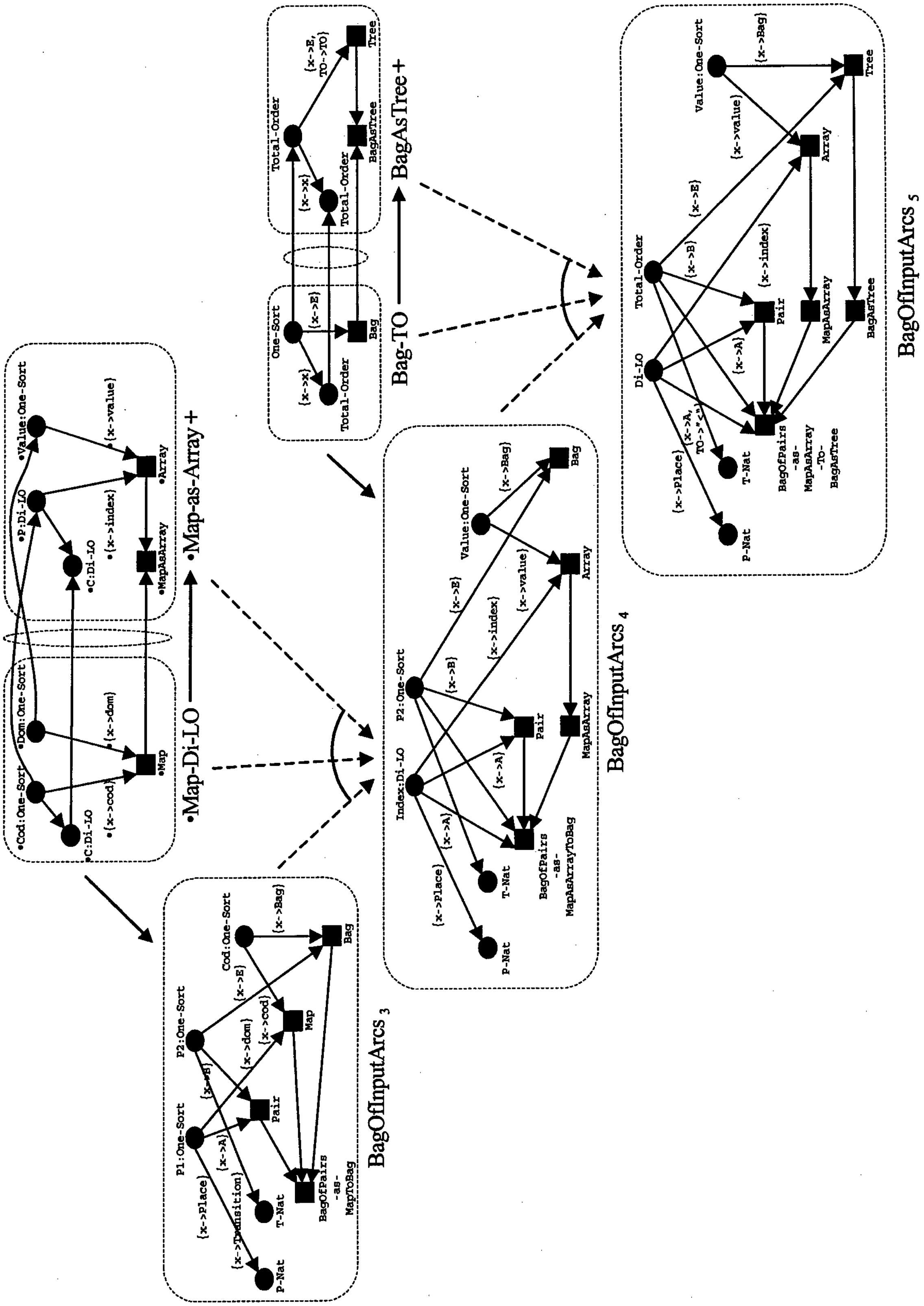


Figure 6-29. Petri Net data structure example II

Because the TransitionAsNat sort is also a discrete linear order, an alternate fourth design decision could have been to refine the Bag of Transitions into an Array of occurrences. In this design choice the array would be indexed by the TransitionAsNat value and the value at that array position would be the count of the number of occurrences in the Bag of that particular index value. The choice as to which fourth design decision to make should be subject to the designer's knowledge of the problem. Since for any given Petri Net the number of transitions in any particular Bag is will in all probability be low, the original design choice is probably best as it will conserve space under those conditions. If the designer had advance knowledge that fully interconnected Petri Nets (with multiple arcs between any given places and transitions) were to be modeled, then the alternate fourth design decision might be better, as it would conserve space and be faster to access under those conditions.

Assuming that one already has arrays and trees implemented in the target programming language, this collection of design decisions is complete. (Except for ensuring that all code generation requirements have been met, see Section C.5.) The abstract requirement a bag of input arcs (a bag of <place, transition> pairs) has, through a series of design decisions and associated additions of design information, been transformed into an array of trees. Even though the entire Petri Net diagram was not depicted during the refinements in Figure 6-28 and Figure 6-29, the four design decisions in those figures are, for the most part, independent of the rest of the Petri Net diagram.

This final example demonstrates that the technique of storing design information as (repartitioned and structure-reduced) diagram interpretations solves the problems and misapplications of *Spec* interpretation and diagram refinement expounded upon in Section 2.5. It involves restructuring design information, parameterized design information and constrained parameterized design information. Although Figure 6-28 and Figure 6-29 did not depict it, the requirement diagram being refined is actually just a sub-diagram of the Petri Net example (Section 5.3.6 and Appendix D). The sequence of refinements could have been made to the much larger Petri Net diagram without encountering any of the problems associated with diagram refinement.

6.5 Contributions and Future Work

Contributions of this chapter include the use of diagram interpretations as a mechanism for representing structured design information, specifically parameterized diagram interpretations and constrained parameterized diagram interpretations. The definitions of these specialized diagram interpretations enable parameterized design information to be represented and applied in a correct manner. For parameterization diagrams in general, they enable the body part to be refined independently of its formal parameter part, as guaranteed by the conservative diagram interpretation morphism between the formal parameter parts and the body parts in the (re)partitioned diagram interpretation. For constraints on the parameter part they enable the application of that design information to ensure that the actual parameter in the requirement diagram satisfies those constraints before the design information can be applied. Constrained parameterized diagram interpretations are a powerful tool, as they enable the refinement of a body part to take advantage of the properties in a particular instantiation by guaranteeing that those properties exist. The different forms of design information are shown to compose, which means that larger chunks of design information can be built up from smaller pieces and applied as a unit instead of piecemeal.

A second contribution of the chapter is the use of pushout diagrams in *dSpec* to apply design information, i.e. (re)partitioned diagram interpretations, to requirement specifications. Using a pushout as the means of applying design information automatically induces the needed diagram morphism between successive versions of the requirement specification and ensures that any preconditions of the design information are satisfied before the design information can be applied.

A third contribution of the chapter is the capability of reducing the structure of the diagram interpretation design information so that it can be more easily generated and applied. While the full structure of parameterized diagram interpretations and constrained parameterized interpretations may be needed to ensure that one is correctly refining the body independent of the formal parameter, that full structure is not needed to represent and apply the design information, as was demonstrated in Section 6.3. (The full structure is needed to verify the design information is correct but that structure is not needed to apply the design information correctly.)

Future work involves investigating other classes of design information that can be represented as a diagram and applied via pushout. An example is the case where the "precondition" structure of the design information involves component specifications and a property defined over an aggregate of the components. The preconditions of the design information cannot be satisfied by the components alone as it is only in the aggregate that one can determine if certain properties hold. A simple example of such a problem involves the school domain, where there exists a set of students and a set of instructors and a relation over their aggregate that ensures that each student has an advisor relation with one of the instructors in the set of instructors. If such a condition is stated in the aggregate, then by applying one of these design decisions the relationship can be embedded in both the student and teacher sorts. The teacher sort can be extended to contain a set of students for which he or she is the advisor and the student sort can be extended to contain the instructor that is his or her advisor. An alternate design decision would be to contain that information in either the teacher sort or the student sort but not both. Yet another design decision would be to create a new sort independent of the teachers and students that contains the relationship.

7 Conclusions

The purpose of this investigation was to develop methods for constructing and refining structured algebraic requirement specifications, and more specifically the representation and application of design information. The current method of constructing requirement diagrams (diagram construction by listing nodes and arcs) does not scale well for large aggregate objects. Another aggregate specification development method, parameterized specifications, results in a specification rather then a structured diagram. Current representations of design information (specifications, morphisms, and interpretations) and methods for design information application (diagram refinement) are not able to represent design information adequately so that it can be applied correctly.

As part of the dissertation research, a category of diagrams and diagram morphisms was developed and applied to algebraic specifications and morphisms that enables the structure of requirement specifications and design information to be dealt with explicitly. A theory of diagram parameterization and instantiation was developed that enables large requirement specifications to be built using a parameter-passing analogy rather than a diagram building analogy. A theory of diagram interpretations was developed that enables structured design information to be correctly represented and applied, including the refinement of parameterized diagrams, restructuring refinements, and establishing preconditions for the application of the design information. Together these innovations solve the problem that current methods have with constructing requirement specifications and correctly representing and applying design information.

The specific contributions of this research are described in Section 7.1, followed by possible future work in Section 7.2.

7.1 Contributions

The first contribution is the formalization of a category of diagrams, dX, in Chapter 3. While diagrams are ubiquitous in category theory and the notion of a diagram as a functor from a shape category to a target category is also common [AHS90], this is the first time that the diagrams themselves have been treated as objects in their own category. The arrows of a diagram category, diagram morphisms, ensure that both the content and the structure of a source diagram are preserved in a target diagram. The formalization of the category *d*X is generic in that it applies to all other categories, X. This generic formalization makes this contribution more applicable than if the category of diagrams had been developed in the framework of a concrete category.

The second contribution is the collection of operations and functors related to the diagram category and its underlying category. The colimit operation of Nice category *d*X diagrams is particularly helpful as it enables an aggregate structure to be developed given a diagram of related diagrams. Even though the full category *d*X has not been proved to be cocomplete, colimits over the Nice *d*X diagrams are sufficient for their use in applying and composing design information in Chapter 6. The functors *Diagramize* and *Colimit* are obvious, but they enable the objects and arrows of category *d*X to be treated as objects and arrows in category X and vice versa. This enables properties over the objects and arrows in category X to be assigned to the associated objects and arrows of category *d*X. The functors are used in Chapter 4 to assign the properties and semantics of category *Spec* objects and arrows to category *dSpec* objects and arrows. The operations *Flatten* and *Partition* relate the diagrams of the categories X and *d*X instead of the individual objects and arrows. They enable category *d*X diagrams to be (re)partitioned and manipulated as needed without losing the underlying structure as a diagram in category X. They are used in Chapter 6 to define necessary properties of diagram interpretations so that they can accurately represent the refinement of parameterized diagrams. Also of note is the definition of an extension morphism in the category *d*X along with the notions of parameterization and instantiation. They are used, along with the operations *Join* and *Fold*, to define the semantics of the diagram statement developed in Chapter 5. The generic incarnation of parameterized diagrams in Chapter 5 is more general than the use to which they have been put in this thesis.

A third contribution is the category *dSpec* developed in Chapter 4 by building upon the diagram theory developed in Chapter 3. The semantics of the objects and arrows of category *dSpec* are formally defined, using the functor *Colimit*, to be the same as the associated category *Spec* objects and arrows. In fact, this formalization has enabled *dSpec* objects and arrows to be treated as *Spec* objects and arrows, which means that much of the theory that is already known about category *Spec* can be directly applied to category *dSpec*. Thus when new properties are attributed to the objects and arrows of category *Spec* they can easily be raised up to be properties of category *dSpec*.

A fourth contribution is the parameterization theory for diagrams that subsumes all current (category theory based) parameterization and instantiation mechanisms for algebraic specifications as described in Section 2.3 and Appendix E. This formalization of parameterization is shown to provide two distinct advantages over other forms of parameterization. The first advantage is that it enables the body being parameterized to be a *dSpec* object (*Spec* diagrams) instead of a single specification. This enables the body to have structure, so to speak, instead of having that structure be collapsed into a single specification. The second advantage is that the definition of a formal parameter includes additional constraints beyond those of being a conservative extension, which is the usual definition of parameterization. These constraints ensure that the formal parameter fully covers the variable part of the body; the defined formal parameter is not *a* formal parameter (i.e. one of many) but *the* formal parameter (i.e. there are no others). This distinction is important when reasoning about the refinement of a body diagram, as the only way to determine if the body refinement has affected its variable parts is to ensure that a formal parameter that fully covers the variable parts still has a conservative extension to the refinement of the body. Also of note is the observation that while formalizations of parameterization in category *Spec* may have many different forms (single parameterization, multi-parameterization, parameterization by a diagram, or the myriad of forms that parameterized diagrams can have), parameterizations when viewed in *dSpec* have only one form, (see Figure 4-10). This may enable automated tools to deal with the endless category *Spec* forms of parameterization by dealing with them on the *dSpec* level as a single form.

A fifth contribution is the use of *Spec* diagrams as reusable extendable objects in a software-engineering context. No longer are specifications, morphisms and interpretations the largest reusable objects for creating *Spec* diagrams. The use of parameterized *Spec* diagrams adds even more expressive power and enables *Spec* diagrams to be created using instantiation, which is undoubtedly a more natural and higher-level diagram development method than the diagram construction method of listing the diagrams nodes and arcs.

A sixth contribution is the development of the syntax and semantics of a diagram statement for creating and instantiating parameterized diagrams. This bridges the gap between the underlying theory and its use in a software-engineering environment. The syntax of the diagram statement is defined in terms of the parameterized diagram theory developed in Section 3.5 and Section 4.2 and is demonstrated to be capable of single parameterization, multi-parameterization and diagram parameterization as well as nested and recursive instantiation. In addition, the parameterized diagram statement is demonstrated to be capable of being partially instantiated so that larger parameterized diagrams can be constructed from smaller parameterized diagrams.

A seventh contribution is the definition of a *dSpec* interpretation and the category *dInterp* that parallels a *Spec* interpretation and the category *Interp* and that forms the basis for representing design information. While a *Spec* interpretation can insure the content of a specification is refined, a *dSpec* interpretation can insure the content and structure are refined. Because of the relationships between categories *Spec*, *dSpec*, *Interp* and *dInterp*, the model semantics of a diagram interpretation is also easily defined. This dissertation argues that a diagram interpretation is a generalization of both an interpretation and diagram refinement. As such , a diagram refinement could be replaced by a diagram interpretation and the latter could provide more refinement capabilities. However, it is better to use diagram interpretations to represent design information and let the diagram colimit mechanism apply the design information.

An eighth contribution is the three classes of design information (simple, parameterized, and constrained parameterized) and the related required diagram interpretation properties described in Section 6.1.1 that ensure that the design information classes are correctly represented and refined. For parameterization diagrams in general, they enable the body part to be refined independently of its formal parameter part as guaranteed by the dInterpretation morphism between the formal parameter parts and the body parts in the (re)partitioned diagram interpretation. For constraints on the parameter part, they ensure that the constraint on the domain parameterized diagram is incorporated into the formal parameter parts of the mediator and codomain diagrams. While notions of the refinement of parameterized specifications have occurred in the literature [Sri97], the notion of a constraint on the parameterization is a new addition to parameterization refinement. Constrained parameterized diagram interpretations are a powerful tool as they enable the refinement of a body part to take advantage of the properties that are present in a particular actual parameter instantiation in the requirement specification. The different forms of design information are shown to compose, which means that larger chunks of design information can be built up from smaller pieces and applied as a unit instead of piecemeal.

A ninth contribution is the application of design information by pushouts in *dSpec*. Once the preconditions of the design information are related to the requirement diagram, the design information can be mechanically applied in a correct manner. While other researchers have developed a means to refine parameterized specifications [Sri97], the application of such design information to a requirement diagram was lacking. Incorporating preconditions as part of the design application process ensures that design information is only applied in situations where it is applicable. Each application of design information results in a new requirements diagram where the design decisions are automatically rippled thorough the diagram. Additionally, the application of design information automatically results in a diagram morphism between the old requirements and the new requirements. The application of design information can be done by a single design decision at a time (with a pushout) or by multiple design decisions at a time (via a colimit). The method of representing and applying design information developed in this thesis is shown to address problems with the current techniques for representing and applying design information (i.e. *Spec* interpretations and diagram refinement).

7.2 Future Work

Several items of future work were discussed in the summaries of the associated chapters. They are summarized here and are followed by other more general observations.

In Chapter 3 future work involves extending the algorithm for colimits over Nice category *d*X diagrams to non-Nice diagrams (those that don't commute and those that contain non-commuting *Spec* diagrams as objects) In Chapter 4 future work involves developing more and better means (mechanical/automatic) for determining if a diagram morphism (and a *Spec* morphism) is a conservative extension. This is of great importance to all forms of algebraic specification parameterization and not limited to diagram parameterization or the research presented here. Related to that is developing a mechanical method for determining if the variable parts of a body diagram are fully covered by the formal parameter. Finally, in Chapter 6 future work involves using diagram interpretations for other classes of design information beyond the classes described in Section 6.1.

In general, more experience is needed in developing and refining requirement specifications using these methods so as to learn the best way to develop, represent, and apply the design information. While this research has provided the underlying theory of representing and applying structured design information and offered some alternatives such as diagram interpretations vs. reduced-structure diagram interpretations, what will work best in practice has not been determined. Futhure research should also involve new specification language syntax (beyond what was developed in Chapter 5 for parameterized diagrams) for developing and applying design information and/or a graphical environment for doing so.

Finally, this dissertation concentrated on the data structure aspects of design. The algorithm choices are equally important. Methods for representing and applying algorithm design information need to be developed and then integrated with the data structure methods in order for a formal transformation software development system to be a viable alternative to traditional methods.

Appendix A. Category Theory

Category theory is an abstract mathematical structure that focuses on the relationships between objects in a collection and not on the objects' internal structure. That is, an object is defined exclusively by its relationships with other objects, where the relationships are referred to as arrows. An arrow between two objects means that the source object's essential properties, from the perspective of the category, are preserved in some way in the target object even though the internal structures of the objects may be different. The focus on property-preserving arrows by category theory makes it perfect for relating algebraic specifications with different internal but similar external properties. Abstracting algebraic specifications and the relationships between them as a category provides a means by which a rich existing logic can be used to describe and prove theories about the refinement of algebraic specifications. The following references were used when developing this Appendix: [Gol84], [AHS90].

A.1 Categories

An axiomatic definition of a category is taken from [Gol84].

Definition A.1. A *category* C consists of the following:

- A collection of things called C-objects
- A collection of things called C-arrows (or C-morphisms)
- Operations assigning to each C-arrow f a C-object dom(f) and a C-object cod(f), where dom and cod are called the domain and codomain objects of the arrow. If A = dom(f) and B = cod(f) then we display this as f:A → B or $A \xrightarrow{f} B$.
- An operation assigning to each pair of C-arrows, $\xrightarrow{f}\xrightarrow{g}$ with cod(f) = dom(g), a C-arrow $g \circ f$: dom(f) → cod(g), the *composite* of f and g, that satisfies the following associative law: Given the configuration $A \xrightarrow{f} B \xrightarrow{g} C \xrightarrow{h} D$ of C-objects and C-arrows then $h \circ (g \circ f) = (h \circ g) \circ f$. See Figure A-1.
- An assignment to each C-object B a C-arrow id_B:B → B, called the identity arrow on B, that satisfies the following Identity law: For any C-arrows f: A → B and g : B → C, $id_B \circ f = f$ and $g \circ id_B = g$.

$$\begin{tikzcd} A \arrow[r, "f"] \arrow[rr, dashed, bend right, "g \circ f"'] & B \arrow[r, "g"] & C \end{tikzcd}$$

Arrow composition

$$\begin{tikzcd} A \arrow[r, "f"] \arrow[d, dashed, "h \circ (g \circ f) = (h \circ g) \circ f"'] \arrow[dr, dashed] & B \arrow[d, "g"] \arrow[dl, dashed] \\ D & C \arrow[l, "h"] \end{tikzcd}$$

Arrow associativity

Figure A-1. Composition and associativity of arrows in a category

Notation: Objects(C) denotes all objects in a category C. Arrows(C) denotes all arrows in a category C. $\text{Arrows}_C(A \rightarrow B)$ denotes all arrows in category C with dom(A) and cod(B).

Example A.2. Sets and the functions over sets form a category, *Set*, where the objects are sets of elements and the arrows are functions that map the elements of one set object to those in another.

Set-objects are sets of elements. *Set*-arrows are triples, <dom, cod, map>, where dom is the domain object set, cod is the codomain object set, and map is a set of pairs (α,β) defining a function that maps domain set elements to codomain set elements.

$$\text{map} \subseteq \{(\alpha,\beta) | \ \alpha \in \text{dom} \wedge \beta \in \text{cod}\}$$
such that for all $\alpha \in$ dom there exists a single $\beta \in$ cod where $(\alpha,\beta) \in$ map.

Identity arrows for *Set*-objects are the triple $<A, A, \{(\alpha,\alpha)| \ \alpha \in A\}>$ where A represents an arbitrary *Set*-object and α an arbitrary element of A. The composition and associativity of functions over sets of elements (and hence arrows in the category *Set*) are well known.

Notation: C-arrows often include domain and codomain information in order to distinguish between similar C-arrows. For example, if a *Set*-arrow were only represented by the element mapping relation then there would be nothing to distinguish between two *Set*-arrows with the same domain object and different codomain objects. The different codomain objects can contain the same image elements as well as some additional (different) elements. Often the domain and codomain information is left off of the arrow as for any given arrow the domain and codomain objects are usually obvious.

Example A.3. Graphs and graph-morphisms form a category, *Graph*, where a graph is an unlabeled, directed, multigraph. The objects of the category are 4-tuples, *Graph*-Object = <N, E, *source*, *target*> where N is a set of nodes, E is a set of arcs, *source*:arc→node returns the source node of an arc and *target*:arc→node returns the target node of an arc. Given *Graph*-objects, G_1 and G_2, a graph-morphism $f{:}G_1 \rightarrow G_2$ is a pair of functions that maps nodes and arcs of G_1 to nodes and arcs of G_2 such that *source* and *target* functions are consistent. For a graph-morphism, $f = <f_{node}, f_{edge}>$, to be consistent, if $G_1 = <N_1, E_1, source_1, target_1>$ and $G_2 = <N_2, E_2, source_2, target_2>$ then for all $e \in E_1$, $source_2(f_{edge}(e)) = f_{node}(source_1(e))$ and $target_2(f_{edge}(e)) = f_{node}(target_1(e))$. Identity arrows, the composition of arrows, and the associativity of composition can be proven by extending the proof for category *Set*. Figure A-2 depicts two graphs and a graph morphism (dashed arrows) between them. Only the nodes are identified in the graph morphism as (in this case) the mapping of arcs is unique. In general there may be zero to many different morphisms between graphs.

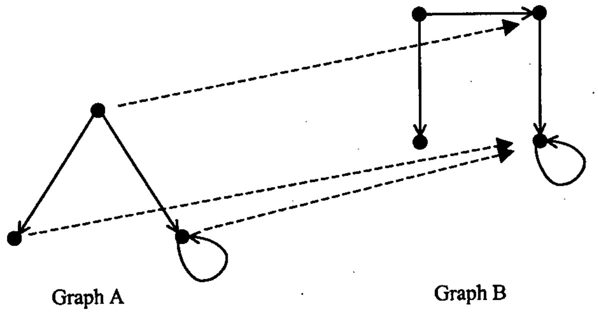


Figure A-2. Graphical depiction of two graphs and a graph morphism between them

Example A.4. Roughly speaking, a shape category is a Graph object that is itself a category: Each node has an arc to itself, and whenever there are adjacent arcs, there is an arc between the source node of the first arc and the target node of the second arc. Thus the nodes and arcs are the objects and arrows of the category. An example of a shape category is depicted in Figure A-3. Chapter 3 formally defines the notion of a shape category, as they are integral to that chapter.

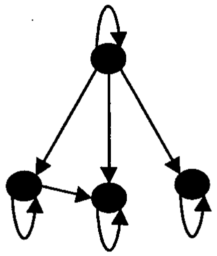


Figure A-3. An example shape category

Example A.5. Any set of elements with a reflexive and transitive binary relation (called a pre-order) among the elements is a category. The elements of the set are the objects of the category and the elements of the pre-order relation among the elements are the arrows. Thus the natural numbers are the objects and the relation "less than or equal to" are the arrows of a particular category. The same set of numbers and the relationship "is divisible by" form a different category. Figure A-4 depicts the "is divisible by" relationship between the natural numbers 1-10, where identity arrows (a number is divisible by itself) and composite arrows are not depicted.

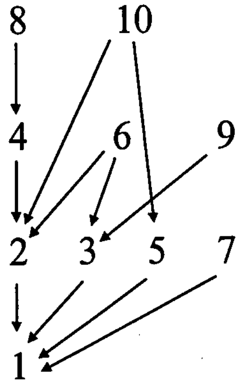


Figure A-4. An example natural numbers category

A.2 Diagrams

Relationships among select objects and arrows of a category can be depicted graphically by using a diagram. The following definition of a diagram is adequate for its use in Appendix A through Appendix C. The definition is expanded upon in Section A.4 and in Chapter 3, the diagram theory chapter.

Definition A.6. A *diagram* in a category C, called a C-diagram, is a directed graph in which vertices are labeled with C-objects and edges are labeled with C-arrows where each edge *f* from vertex η_1 to vertex η_2 has an associated C-arrow f:A→ B, where A is the C-object associated with η_1 and B is the C-object associated with η_2.

Notation: Because there are many arrows and objects in a given category one normally depicts in a diagram figure only representations of those objects and arrows that are of interest. The identity arrow of an object and some of the arrow compositions are not depicted in a diagram because the depiction becomes too unwieldy, but they exist all the same.

Notation: Typically a node in a diagram and its associated object have the same name. If an object is depicted more than once in a diagram, then the nodes are given distinct names.

Diagrams are often used to state and prove properties (as shown in Figure A-1) of categorical constructions. Following a sequence of edges represents arrow composition; the composite arrows need not be depicted, as they always exist. A walk through the vertices and edges of a C-diagram represents a unique composite arrow from the source object of the walk to the target object of the walk. If the composite arrows of two distinct walks that start and end at the same pair of vertices are equivalent, then the walks are said to commute. We say the C-diagram commutes to imply this equivalency for all such walks in the diagram.

Definition A.7. A C-diagram *commutes* if for all vertices A and B, all the walks from A to B yield, by arrow composition in category C, equal C-arrows with domain A and codomain B.

An example of a non-commuting diagram in category *Set* is depicted in Figure A-5; Arrows *f* and *g* have the same domain and codomain objects yet they are not equivalent. The compositions $f \circ h$ and $g \circ h$ are equivalent, however, as $(f \circ h) = (g \circ h) = \mathrm{Z}\to 3$. Neither of these compositions are equivalent to arrow *i*.

$$\begin{tikzcd}
\{Z\} \arrow[r, "{h = Z\to C}"] \arrow[rr, bend right=40, "{i = Z\to 1}"'] & \{A,B,C\} \arrow[r, bend left, "{f = A\to 1, B\to 2, C\to 3}"] \arrow[r, bend right, "{g = A\to 2, B\to 1, C\to 3}"'] & \{1,2,3\}
\end{tikzcd}$$

Figure A-5. Non commuting diagram in category *Set*

A.3 Arrow properties

Some objects in a category with different internal structure can be classified as being abstractly the same object within a category by virtue of having the same external structure. Since category arrows relate the external structure (essential properties) of objects, one can define what it means to be abstractly the same based on the arrows between objects in a category.

Definition A.8. An *isomorphism* is a C-arrow f: A→B for which there exists a C-arrow g:B→A such that $g \circ f = \mathrm{id}_A$ and $f \circ g = \mathrm{id}_B$. The C-arrow *g* is uniquely defined [AHS90] and called the inverse of *f* and is denoted f^{-1}. If there exists a C-arrow f: A→B that is an isomorphism, then A and B are called *isomorphic* C-objects and we write $A \cong B$.

The concept of an isomorphism is important as it identifies objects that have the same structure (from the perspective of the category). This structure is the essential property that the arrows of a category preserve when relating objects via the arrows. Many categorical constructions are only defined up to isomorphism. There may be many isomorphic objects that fit a given definition and we typically pick (or construct) one of them to represent them all.

Objects in the category *Set* are isomorphic if they have the same cardinal number. Sets with the same cardinal number have at least one bijective function between them. The inverse of a *Set*-arrow with a bijective function swaps the domain and codomain and reverses the map relation (the inverse of the *Set*-arrow <A, B, {α→β, μ→ν, etc}> is <B, A, {β→α, ν→μ, etc}>). Any set can abstractly replace any other set with the same cardinal number. What is important (from the perspective of the category *Set*) is the number of elements in the set, not the arbitrary names or internal structures of the elements. For example, the properties of the actual elements of the isomorphic sets {1,2,3} and {a,b,c} are immaterial from the perspective of the category *Set.*

An arrow can also have the effect of mapping the structure of one object to another in a manner that is similar to injective or surjective functions in between sets. These arrow properties are called monic and epic, respectively.

Definition A.9. A C-arrow *f*: A→B is *monic* (or a *monomorphism*) if for any arrows *g*:C→A and *h*:C→A

$$\begin{tikzcd} C \arrow[r, shift left, "g"] \arrow[r, shift right, "h"'] & A \arrow[r, "f"] & B \end{tikzcd}$$

the equality $f \circ g = f \circ \mathrm{h}$ implies $g = \mathrm{h}$.

Definition A.10. A C-arrow *f*: A→B is *epic* (or an *epimorphism*) if for any arrows *g*:B→C and *h*:B→C

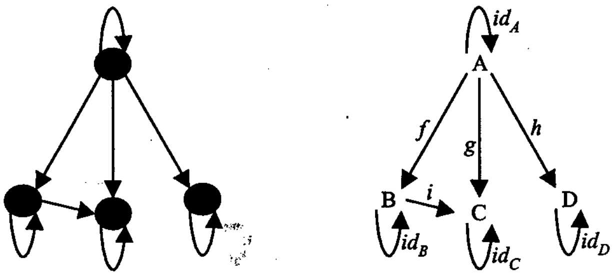
$$\begin{tikzcd} A \arrow[r, "f"] & B \arrow[r, shift left, "g"] \arrow[r, shift right, "h"'] & C \end{tikzcd}$$

the equality $g \circ f = h \circ f$ implies $g = h$.

Monic arrows in *Set* are injective functions and epic arrows in *Set* are surjections. In the category *Set*, if an arrow is both a monomorphism and an epimorphism it is an isomorphism. This is not the case for all categories. Monic arrows are important in that the structure of the domain object is not combined or collapsed somehow in the structure of the codomain object. Epic arrows are important in that the domain object completely covers the codomain object, there is no additional (external) structure in the codomain object.

A.4 Functors

Since a diagram is a "window" into the objects and arrows of a particular category, we know that all of the properties (identity arrows and composition of arrows) of a category apply to the objects and arrows in a particular diagram whether they are depicted or not. Thus another way to view a C-diagram is as a particular relationship between a shape category and the category C; see Figure A-6, where there is an obvious mapping of the objects (nodes) and arrows (arcs) of the depicted Shape category to the selected objects {A,B,C,D} and arrows $\{f, g, h, i, id_A, id_B, id_C, id_D\}$ of category C



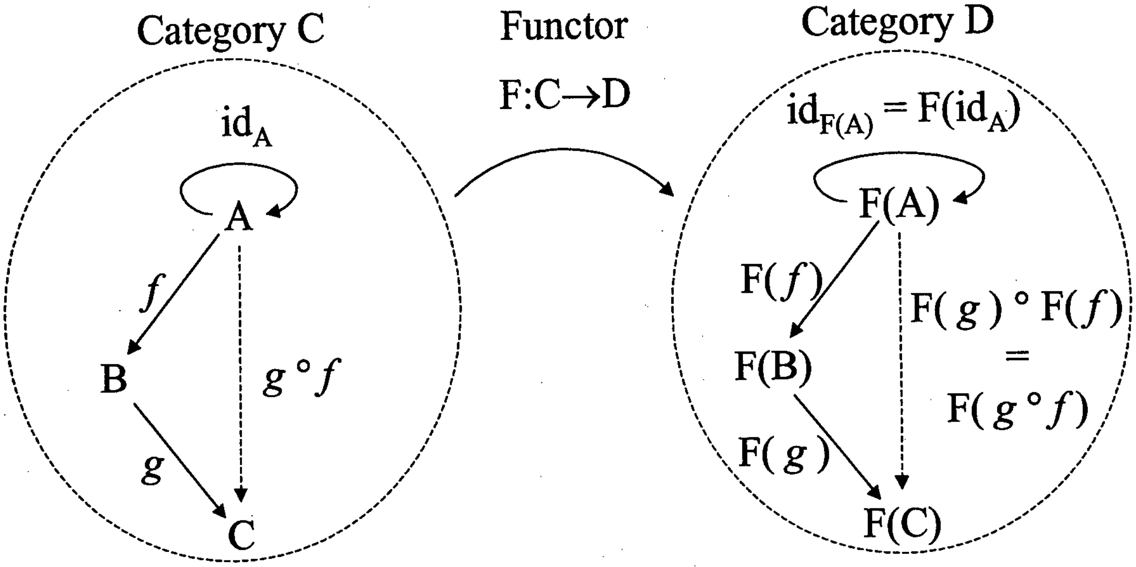
a. Shape Category

b. Diagram of objects and arrows in Category C

Figure A-6. A shape category and a diagram of objects and arrows of a particular category

When two categories are related in a way that preserves the object and arrow structure of the source category in the target category, that relationship is called a functor. Functors map all objects and arrows in one category to those of another such that identity arrows and arrow compositions are preserved.

Definition A.11. A *functor* F from a category C to a category D is a pair of functions $F = <F_{Object}, F_{Arrow}>$ that assigns to each C-object A a D-object $F_{Object}(A)$ and to each C-arrow $f: A \rightarrow B$ a D-arrow $F_{Arrow}(f): F_{Object}(A) \rightarrow F_{Object}(B)$ such that the identity arrows and composite arrows of category C remain identity arrows and composite arrows in category D, as depicted in Figure A-7.



$F_{Arrow}(id_A) = id_{F_{Object}(A)}$ for all C-objects A, and

$F_{Arrow}(g \circ f) = F_{Arrow}(g) \circ F_{Arrow}(f)$ whenever $g \circ f$ is defined in C.

Figure A-7. A Functor preserves identity and composition arrows

Example A.12. A C-diagram can be thought of as the functor between a particular shape category and the category C. If G is the shape category depicted in Figure A-6(a) then pair of functions that assigns each object (node) in G to a C-object and each arrow (arc) in G to a C-arrow such that identity arrows and arrow compositions are preserved is a functor. This notion of a diagram as a functor is integral to Chapter 3 and is further explained there.

Categories and the functors between them are themselves a category, *Cat* [AHS90]. This can cause some "level" confusion, as on one level a functor is between categories and on the *Cat* level it is an arrow between objects.

A.5 Subcategory

Subcategories contain a subset of the objects and arrows of another category. A subcategory is defined in [AHS90] as

Definition A.13. A category C is a *subcategory* of category D iff

- Objects(C) ⊆ Objects(D)
- For each α, β ∈ Objects(C), $Arrows_C(\alpha \to \beta) \subseteq Arrows_D(\alpha \to \beta)$
- For each α ∈ Objects(C), the C identity arrow of α is the D identity arrow
- For each α, β ∈ Objects(C), C composition $\alpha \circ_C \beta$ is a restriction of D composition $\alpha \circ_D \beta$.

Definition A.14. A category C is a *full subcategory* of category D iff

- C is a subcategory of D
- For each $\alpha, \beta \in$ Objects(C), $\text{Arrows}_C(\alpha\rightarrow\beta) = \text{Arrows}_D(\alpha\rightarrow\beta)$

Since a subcategory is based upon another category one does not need to re-prove associativity, composition and identity properties of arrows. By specifying its object class as a subset of Objects(D), we can identify a full subcategory of D. We know this full subcategory is a category as it has all the relevant arrows (including composition and identity) of its subset of D objects and that these arrows (in D) have all the appropriate categorical properties. With subcategories that are not full, we must be sure that all the relevant arrows are included.

A.6 Pushouts, Colimits

A pushout, and the more general colimit, are category constructions that can be used to determine a unique (up to isomorphism) "minimal" object for a given diagram. A pushout is the name given to a colimit over diagrams of the form B1← A →B2.

Definition A.15. In any category, X, given a diagram B1← A →B2 where an object A has arrows *f*:A→B1 and *g*:A→B2, the *pushout* of the diagram is an object C and arrows *f'*:B2→C and *g'*:B1→C where $g' \circ f = f' \circ g$ and where for any other C-arrows *h*:B2→D and *j*:B1→D such that $j \circ f = h \circ g$ there is a unique arrow *k*:C→D that forms a commuting diagram, see Figure A-8. The constructed arrows that form the pushout square are also unique. Arrow *f'* is referred to as the pushout of *f* along *g* and arrow *g'* is referred to as the pushout of *g* along *f*.

$$\begin{tikzcd}
A \arrow[r, "f"] \arrow[d, "g"'] & B1 \\
B2 &
\end{tikzcd}$$

a. Pushout base

$$\begin{tikzcd}
A \arrow[r, "f"] \arrow[d, "g"'] & B1 \arrow[d, dashed, "g'"'] \\
B2 \arrow[r, dashed, "f'"] & C
\end{tikzcd}$$

b. Pushout constuction

$$\begin{tikzcd}
A \arrow[r, "f"] \arrow[d, "g"'] & B1 \arrow[d, dashed, "g'"'] \arrow[rdd, bend left, "j"] & \\
B2 \arrow[r, dashed, "f'"] \arrow[rrd, bend right, "h"] & C \arrow[rd, "k"] & \\
 & & D
\end{tikzcd}$$

c. Pushout is minimal

Figure A-8. Definition of pushout

The pushout object C, in Figure A-8, is related to each object in the pushout base diagram as there exist arrows from those objects to the pushout object. The object C is minimal in that any other object that forms a commuting square with the pushout base must have an arrow to it from the pushout object.

An example pushout in the category *Set* is depicted in Figure A-9. The pushout object of diagram $\{1,2,3\} \leftarrow \{A,B\} \rightarrow \{7,8,9\}$ is object $\{W,X,Y,Z\}$ and the arrows to that object. The actual elements in the pushout object (W,X,Y,Z) are arbitrary; any Set with four elements could serve as the pushout set object as it would be isomorphic to the one depicted. The interested reader can construct commuting squares with set objects that have 1 through 5 elements and determine if the given pushout object in Figure A-9 has a unique arrow to it.

$$\begin{tikzcd}
 & \{A,B\} \arrow[dl, "g = A\rightarrow 1,\, B\rightarrow 2"'] \arrow[dr, "f\ = A\rightarrow 7,\, B\rightarrow 7"] & \\
\{1,2,3\} \arrow[dr, dashed, "f' = 1\rightarrow W,\, 2\rightarrow W,\, 3\rightarrow X\}"'] & & \{7,8,9\} \arrow[dl, dashed, "g' = 7\rightarrow W,\, 8\rightarrow Y,\, 9\rightarrow Z\}"] \\
 & \{W,X,Y,Z\} &
\end{tikzcd}$$

Figure A-9. Example pushout in *Set*

A pushout is a specific example of the colimit operation that defines the "minimal" object for an arbitrary diagram in category.

Definition A.16. A *cocone* from a diagram $\mathcal{D}$ to an object C is a collection of arrows $\{f_i\colon D_i \rightarrow C \mid D_i \in \mathcal{D}\}$ such that for any arrow $g{:}D_i \rightarrow D_j$ in $\mathcal{D}$, $f_i = f_j \circ g$. That is, diagram in Figure A-10commutes for all such diagrams involving objects and arrows of $\mathcal{D}$ and the collection of arrows to C.

$$\begin{tikzcd}
D_i \arrow[rr, "g"] \arrow[dr, "f_i"'] & & D_j \arrow[dl, "f_j"] \\
 & C &
\end{tikzcd}$$

Figure A-10. Definition of a cocone

Definition A.17. A *colimit* for a diagram $\mathcal{D}$ is an object C along with a cocone from $\mathcal{D}$ to C such that for any other cocone to an object C', there is a unique arrow from f: C→C' such that for every object D_i in $\mathcal{D}$, $f_i' = f \circ f_i$. That is, the diagram in Figure A-11 commutes for all such diagrams involving the objects of $\mathcal{D}$, their cocone arrows to C and C' and the unique arrow f:C→C'.

$$\begin{tikzcd}
& D_i \arrow[dl, "f_i"'] \arrow[dr, "f_i'"] & \\
C \arrow[rr, "f"'] & & C'
\end{tikzcd}$$

Figure A-11. Definition of a colimit

Some categories do not have colimit objects for arbitrary diagrams.

Definition A.18. A category is said to be *cocomplete* it if has all colimits.

Proposition A.19. A category that has an initial object (an object that has an arrow to every object in the category) and all pushouts is cocomplete

Proof: This is a well-known theorem of category theory [Gol84].

Appendix B. Specifications and Models

This appendix presents an algebraic-based view of a software requirement specification. Algebraic specification was introduced in the mid-1970s as a formal technique for representing and reasoning about data structures such as sets, lists, strings, etc., in an implementation-independent manner [EMCO92]. The desired interface and properties of an abstract data type (ADT) can be formally expressed using an algebraic specification and can be reasoned about independent of any concrete implementations of the ADT, as any implementation of the specification must have (at least) those properties. This approach to representing and reasoning about data structures is similar to the way that abstract mathematical structures such as groups, rings and fields are described and reasoned about in modern algebra.

The requirements for an abstract data type can be specified using an algebraic specification by defining the ADT's interface (naming the types and the operations involved) and listing the characteristic properties of that ADT (logical characteristics over the interface elements). As an example, the equation "Insert(e,S) = Insert(e,(insert(e,S))" indicates that inserting something twice is equivalent to inserting something once which is a property of the Set ADT but not the List or Bag ADT. When the logical characteristics are limited to the observable interface, then the properties of the abstract data type are expressed without reference to its internal implementation details. Any programming language implementation that satisfies the properties of the algebraic specification is a valid implementation of the specification. Because of this, algebraic specifications provide an excellent means for describing the requirements for a system while not constraining the designer to implement it a specific way.

One of the meanings of a specification is the class of *models* associated with that specification. A model (often called an algebra) consists of a finite number of sets of elements and a finite number of typed functions over the elements of the sets. If the sorts and operations of a specification are associated with the sets of elements and functions of a model in such a way that all of the logical properties in the specification are satisfied, then the functions and sets of elements are said to be a model of the specification. Any collection of Fortran or Ada functions can be abstracted as model where an abstract function (input and output relations only, no algorithms) represents a concrete programming language function and a set of elements represents the range of values of a particular programming language type. Thus an algebraic specification can be used to specify a class of models and in doing so indirectly also specifies a class of programming language implementations that are concrete versions of the models.

A design decision can be viewed as a collection of additional properties that an implementation must have beyond those made necessary by the purely functional requirements. An algorithm or data structure choice is an example of a design decision. The process of taking an abstract requirement specification into a concrete design specification is called refinement. Refinement is finished when a programming language implementation can be derived from the design specification.

Section B.1 of this appendix describes algebraic specifications and Section B.2 covers models of algebraic specifications. For a more detailed overview of algebraic specifications, see [Wir90]. For a different category-theory-based overview of algebraic specifications, see [Sri90].

B.1 Algebraic Specifications

In this section algebraic specifications are defined by first defining the components of a specification: a signature of sorts and operations, and axioms that are logical sentences over the signature. This section also describes how the components of a specification relate to each other: how axioms are related to a signature, how signature morphisms relate the signatures of two different specifications, how signature morphisms can be used to translate axioms, and how an entailment system relates set of axioms. Finally it defines specifications and specification morphisms that ensure that the properties of the source specification are present in the target.

B.1.1 Signatures

The interface of an abstract data type consists of a set of data types and a set of functions over those data types. These types and functions can be abstractly described using a signature consisting of a set of sorts and a set of operations over those sorts. A signature is only an abstract representation of the interface and does not contain any information that describes the functionality of the abstract data type.

Definition B.1. A *signature* $\Sigma = <S, \Omega>$ consists of a set S of sort symbols and an indexed family $\Omega_{w,s}$ of sorted operation names over the sorts in S, where for all operations $f \in \Omega_{w,s}$ with $w = s_1, s_2, \ldots s_n$, the set $\{s_1, s_2, \ldots s_n, s\}$ must be a subset of S.

Each signature $\Sigma = <S, \Omega>$ has two functions, *Sorts* and *Operations*, where *Sorts*(Σ) = S and *Operations*(Σ) = Ω. In addition, we write f: $s_1, s_2, \ldots s_n \rightarrow s$ to indicate the sort type of each $f \in \Omega$ and we use the sequence $[s_1, s_2, \ldots s_n, s]$ to indicate the rank of f. If rank(f) = [s] (the singleton sequence) then f is called a constant of sort s.

Example B.2. A simple signature, SET, for the set abstract data type is the pair of sets:

<{set, E, boolean, nat}, {Empty: →set, insert: E, set→set, in: E, set→boolean, size: set→nat}>

Example B.3. A simple signature, LIST, for the list abstract data type is the pair of sets:

<{list, E, boolean, nat}, {Empty:→list, append: E, list→list, in: E, list→boolean, length: list→nat}>

B.1.2 Signature morphisms

Two signatures can be related by a signature morphism if the sorts and operations of one can be mapped to the other in a way that preserves the ranks of the operations. Just as a signature cannot describe the "functionality" of sorts and operations, a signature morphism cannot indicate that the sorts and operations being related have similar "functionality".

Definition B.4. A *signature morphism* σ: A → B from signature A to signature B is a pair of functions $\sigma = <\sigma_S, \sigma_\Omega>$ where for all $s \in$ *Sorts*(A) and $f \in$ *Operations*(A) there must exist a $s' \in$ *Sorts*(B) and $f' \in$ *Operations*(B) respectively, such that $\sigma_S(s) = s'$ and $\sigma_\Omega(f) = f'$ and where the rank of f' is $[\sigma_S(s_1), \sigma_S(s_2), \ldots, \sigma_S(s_n), \sigma_S(s)]$ when the rank of f is $[s_1, s_2, \ldots, s_n, s]$.

Example B.5. The pair of relations: <{set→list, E→E, boolean→boolean, nat→nat}, {Empty→Empty, insert→append, in→in, size→length}> defines a signature morphism SET-to-LIST: SET→LIST.

Notation: In order to make it easier to read signature morphisms, the mappings between sorts and operations with the same names are normally not depicted and the two sets (which are usually disjoint) are merged. Using this syntax, the SET-to-LIST signature morphism can be written more succinctly as {set→list, insert→append, size→length}.

Proposition B.6. Signatures and Signature morphisms form a category, *Sign*, where the objects are signatures and the arrows are signature morphisms.

Proof: The identity arrow for an object in Sign maps the sorts and operations of a signature to themselves in the same manner as the identity arrows in category *Set*. The composition of arrows $\sigma = <\sigma_S, \sigma_\Omega>$ from A to B and $\sigma' = <\sigma_S', \sigma_\Omega'>$ from B to C is $\sigma' \circ \sigma = <\sigma_S' \circ \sigma_S, \sigma_\Omega' \circ \sigma_\Omega>$, which is associative. □

The important parts of a signature from the perspective of the category *Sign* are the number of sorts, number of operations, and ranks of the operations. The signature morphism in Example B.5 is an isomorphism as the signature morphism {list→set, append→insert, length→size} is the inverse of the one given in the example. (The two signature morphisms compose to yield either id_{SET} or id_{LIST}.)

Monomorphisms in *Sign* mean the domain sorts and operations are mapped one-to-one to codomain sorts and operations. There is no combining or collapsing where a single codomain sort or operation is used to represent multiple domain sorts or operations. Epimorphisms in Sign mean the domain sorts and operations cover the codomain sorts and operations. There are no additional codomain sorts and operations that do not have an associated domain sort or operations. An isomorphism between *Sign*-objects can be viewed as a signature renaming as the sort and operation names may change but the size and the rank relations of the signature remains the same.

B.1.3 Axioms

In order to specify (and hence distinguish between) different abstract data types in a way that does not overly restrict implementation choices one must be able to describe their properties independent of the internals of any particular implementation. Logical predicates over the signature of the abstract data type give us a way to describe properties of the abstract data type over its interface while not prescribing a specific internal structure.

Definition B.7. A *ground term*, for a signature $\Sigma = <S, \Omega>$, is an element of $\mathcal{G}_\Sigma$, where $\mathcal{G}_\Sigma$ is the set of sort-indexed terms generated using only the signature constants and signature operators over those constants. $\mathcal{G}_\Sigma$ is a set of sort-indexed sets, $\mathcal{G}_\Sigma \overset{\text{def}}{=} \{\mathcal{G}_{\Sigma,s} \mid s \in S\}$ where $\mathcal{G}_{\Sigma,s}$ is defined inductively as follows:

- if c is a constant operator of sort $s \in S$, then c is a ground term of sort s and $c \in \mathcal{G}_{\Sigma,s}$
- if $f \in \Omega$ is an operation with rank $[s_1, s_2, \ldots s_n, s]$ and $t_1, t_2, \ldots t_n$, are ground terms in $\mathcal{G}_{\Sigma,s1}, \mathcal{G}_{\Sigma,s2}, \ldots, \mathcal{G}_{\Sigma,sn,}$ respectively, then $f(t_1, t_2, \ldots t_n)$ is a ground term of sort s and $f(t_1, t_2, \ldots t_n) \in \mathcal{G}_{\Sigma,s}$.

Example B.8. The ground terms for the signature SET from Example B.2 form the set $\mathcal{G}_{SET}$ ={Empty, Size(Empty)}. The SET operations *Insert* and *In* both require a ground term of sort E as a "parameter" before they could be considered ground terms, therefore $\mathcal{G}_{\text{SET,E}}$ is empty because sort E has no constants defined.

Definition B.9. A *term*, for a signature $\Sigma = \langle S, \Omega \rangle$, is an element of $\mathcal{T}_{\Sigma}(V)$, where $\mathcal{T}_{\Sigma}(V)$ is the set of sort-indexed terms $\mathcal{T}_{\Sigma,s}(V)$ generated from a set of sorted variables $V = \{V_s \mid s \in S\}$ and signature constants, and the signature operators over those constants and variables.

$\mathcal{T}_{\Sigma}(V) \stackrel{\text{def}}{=} \{\mathcal{T}_{\Sigma,s}(V) \mid s \in S\}$ where $\mathcal{T}_{\Sigma,s}(V)$ is defined inductively as follows:

- if v is a variable of sort $s \in S$, then v is a term of sort s and $v \in \mathcal{T}_{\Sigma,s}(V)$
- if c is a constant operator of sort $s \in S$, then c is a term of sort s and $c \in \mathcal{T}_{\Sigma,s}(V)$
- if $f \in \Omega$ is an operation with rank $[s_1, s_2, \ldots s_n, s]$ and $t_1, t_2, \ldots t_n$, are terms in $\mathcal{T}_{\Sigma,s1}(V)$, $\mathcal{T}_{\Sigma,s2}(V)$, …, $\mathcal{T}_{\Sigma,sn}(V)$, respectively, then $f(t_1, t_2, \ldots t_n)$ is a term of sort s and $f(t_1, t_2, \ldots t_n) \in \mathcal{T}_{\Sigma,s}(V)$

Example B.10. If we consider e to be a variable of sort E then for signature SET from Example B.2

$\mathcal{T}_{\text{SET}}(\{e\})$ = {Empty, Insert(e,Empty), Insert(e,Insert(e,Empty)), Insert(e,Insert(e,Insert(e,Empty))),…, In(e,Empty), In(e,Insert(e,Empty)), In(e,Insert(e,Insert(e,Empty))), …, Size(Empty), Size(Insert(e,Empty)), Size(Insert(e,Insert(e,Empty))), …}.

The sets of terms and ground terms over a signature are related in that the set of terms using no variables is the set of ground terms, i.e. $\mathcal{T}_{\Sigma}(\{\}) = \mathcal{G}_{\Sigma}$.

An axiom is a sentence over the terms of a given signature and additional logical predicates. (Such as equals, and, or, not, implies, iff ($=, \wedge, \vee, \neg, \Rightarrow, \Leftrightarrow$) and the universal and existential quantifiers ($\forall, \exists$) and the bound variables they introduce.) The definitions for axiom sentences (terms over logical predicates and quantifiers) are similar to the inductive definitions given in Definition B.7 and Definition B.9. The exact definition for the axiom sentences is dependent on the particular logic one wishes to use (equational logic, conditional equational logic, first order logic, etc.) for describing abstract data type properties. There are many different kinds of restrictions one can place on axiom sentences in terms of permitted logical predicates, quantifiers and syntactic sentence forms so that the axiom sentences are associated with a particular logic.

The following list demonstrates three of the most common logics used in algebraic specification:

▪ Equational logic	size(empty) = 0
	size(prepend(x, a-seq)) = 1 + size(a-seq)
▪ Conditional Equational logic	in(x, a-set) => (size(insert(x, a-set)) = size(a-set))
	not(in(x, a-set)) => (size(insert(x, a-set)) = 1 + size(a-set))
▪ First-Order logic	size(a-set) = 0 <=> ∀ x.element | not(in(x, a-set))

Restricting the logic affects the expressiveness in terms of what can be said as well as how easy it can be said. Restricting the logic may enable a more efficient reasoning system or have other interesting properties. For example a term rewriting system that enables a specification to be executed or interpreted directly requires equational logic or conditional equational logic [Klop92]. If computability is the primary concern then equational logic with hidden functions has been shown to be sufficient for describing all computable algebras [MG84]. However, the need for hidden functions means that an internal structure is specified along with the external behavior. In addition, some properties that are simple to describe using higher order logic are quite complex when described using a lower order logic, such as equational logic. On the other hand, higher order logics may not have complete proof systems, which makes proving properties about sets of axiom sentences more difficult.

Rather then relying on a particular logic and defining exactly what is meant by an axiom, we assume that for every signature Σ there exists a well-formed and well-sorted set of sentences over that signature called Σ-sentences. The axioms describing an abstract data type with a signature of Σ are a subset of the Σ-sentences. The collection of Σ-sentences for all possible signatures and the translation mappings between these sets of Σ-sentences form a subcategory of the category *Set*.

B.1.4 Axiom translation over a signature

Two categories, such as *Sign* and *Set*, can be related by an operation that maps objects in one category to those in another in a way that preserves the arrows between the objects. If there is a morphism between Σ and Σ' in the category *Sign* it makes sense that there be a relationship between the sets of Σ-sentences and Σ'-sentences. A functor between the *Sign* category and the *Set* category captures that relationship.

Proposition B.11. The pair of functions < Sen_{Object}, Sen_{Arrow} > is a functor *Sen*: *Sign* → *Set* where Sen_{Object} maps each signature object Σ in *Sign* into the Σ-sentences object in *Set* and Sen_{Arrow} maps the signature morphism arrow σ:A→B in *Sign* into the function that translates A-sentences to B-sentences in *Set* in a manner that is consistent with the signature mapping.

Proof: For any id_Σ in *Sign*, $Sen_{\text{Arrow}}(\text{id}_\Sigma)$ translates Σ-sentences to Σ-sentences therefore $Sen_{\text{Arrow}}(\text{id}_\Sigma) = \text{id}_{Sen_{\text{Object}}(\Sigma)}$ and *Sen* preserves identity arrows. If *f* and *g* are composable signature morphisms in *Sign* then $Sen_{\text{Arrow}}(g \circ f)$ translates sentences from the domain of *f* to the codomain of *g*. This is equal to $Sen_{\text{Arrow}}(g) \circ Sen_{\text{Arrow}}(f)$, which is the composition of two translation functions. □

Example B.12. Sen_{Object}(SET) and Sen_{Object}(LIST) map the *Sign*-objects SET and LIST (from Example B.2 and Example B.3) to their sets of well-formed sentences, SET-sentences and LIST-sentences, in the category *Set*. The *Sign*-arrow SET-to-LIST: SET → LIST is mapped by Sen_{Arrow} to a *Set*-arrow that translates SET-sentences to LIST-sentences, i.e. Sen_{Arrow}(SET-to-LIST):SET-sentences→LIST-sentences. Thus the *Set*-arrow Sen_{Arrow}(SET-to-LIST), when used as a function, maps the *Sen*(SET) element "∀ *s*:set, *e*:E | in(*e*,insert(*e*,*s*))" to the *Sen*(LIST) element "∀ *s*:list, *e*:E | in(*e*,append(*e*,*s*))".

B.1.5 Entailment

In order to prove properties about an abstract data type based on existing properties it is important to be able to prove relationships between sets of sentences in a given language. A proof calculus for the particular logic underlying the Σ-sentences is needed (such as natural deduction being the proof calculus for first order predicate logic [Man74]). Rather then limiting ourselves to a particular logic or proof calculus we define an entailment system that abstracts away from the particulars of any specific logic and proof calculus. An entailment relation is used to assert that a sentence is provable from a set of sentences but does not prescribe the logic underlying the sentences or the calculus by which the sentence is proved. The following definition for an entailment system is based on [Mes89].

Definition B.13. An *entailment system* is a triple $\mathcal{E}$= <*Sign*, *Sen*, ⊢> with *Sign* and *Sen* defined as above and ⊢ a family of functions associated with each Σ in *Sign* such that each $\vdash_\Sigma:\mathcal{P}(Sen(\Sigma)) \rightarrow Sen(\Sigma)$ called Σ-entailment has the following properties:

- Reflexivity: for any $\alpha \in Sen(\Sigma)$, $\{\alpha\} \vdash_\Sigma \alpha$;

  If we assume a sentence we can prove it
- Monotonicity: if $\Gamma \vdash_\Sigma \alpha$ and $\Gamma \subseteq \Gamma'$ then $\Gamma' \vdash_\Sigma \alpha$;

  If a sentence can be proved, then it can also be proved with more assumptions
- Transitivity: if $\Gamma \vdash_\Sigma \alpha$ and $\Gamma \cup \{\alpha\} \vdash_\Sigma \beta$ then $\Gamma \vdash_\Sigma \beta$;

  Using a proved sentence as an additional assumption cannot enable more things to be proved (although in any given proof calculus it may make the proof "easier")
- $\vdash$-translation: if $\Gamma \vdash_\Sigma \alpha$, then for any $H:\Sigma \rightarrow \Sigma'$ in *Sign*, $H(\Gamma) \vdash_{\Sigma'} H(\alpha)$where $H = Sen_{\mathrm{Arrow}}(H)$;

  The entailment relation is consistent under translation from one signature to another

B.1.6 Specifications

A signature Σ can describe the interface signature of an abstract data type. A set of axioms, $Ax \subseteq Sen(\Sigma)$, can describe the abstract properties (functionality) of an abstract data type. A specification combines a signature and a set of axioms over that signature. Together they can describe the interface and the properties that any implementation of the specification must have.

Definition B.14. A *specification* $A = <\Sigma, Ax>$ (or $<S, \Omega, Ax>$) consists of a signature Σ and a set of axioms $Ax \subseteq Sen(\Sigma)$ where *Signature*(A) = Σ, *Sorts*(A) = S, *Operations*(A) = Ω, and *Axioms*(A) = Ax.

Example B.15. A simple specification for the Set Abstract Data type is <{set, E, nat, boolean},

{Empty:→set, Insert: E, set→set, In: E, set→boolean, Size:set → nat},

{"$\forall$ *e*:E not(in(*e*,Empty))", "$\forall$ *s*:Set, *a*,*b*:E in(*a*,insert(*b*,*s*)) $\Rightarrow$ ((*a*=*b*) $\vee$ in(*a*,*s*))", etc.}>.

Notation: In order to present a specification in a more readable fashion, we adopt the syntax in Figure B-1. This syntax is a slight adaptation of the Specware language [SLM98]. Note that the sorts Boolean and Nat are fully defined and implicit in the underlying logic. Note also that each axiom is (individually) quantified by the single "`Forall`" context clause:

```
Spec Set is
  sort Set, E
  op Empty:          -> Set
  op Insert: E, Set -> Set
  op In:     E, Set -> Boolean
  op Size:   Set    -> Nat

  Forall s:Set, e,e1,e2:E
  Ax not(In(e,Empty))
  Ax in(e1,Insert(e2,s)) ⇒ ((e1=e2) ∨ In(e1,s))
  Ax Size(Empty) = 0
  Ax Size(Insert(e,s)) = if In(e,s) then Size(s)
                                    else Size(s) + 1
end-spec
```

Figure B-1. Simple Set specification

A distinction is made between the axioms that are part of a specification and the Σ-sentences that can be proved from those axioms. If specification A = <Σ, Ax> and Ax $\vdash_\Sigma \beta$ where $\beta \in \mathrm{Sen}(\Sigma)$ then we consider β to be a theorem of specification A. All axioms of a specification are also theorems of the specification because of the reflexivity and the monotonicity of $\vdash_\Sigma$.

Definition B.16. The *closure* of the set of axioms, Ax, for a specification A = <Σ, Ax> with respect to $\vdash_\Sigma$ is the set $\mathrm{Ax}^* = \{\alpha | \mathrm{Ax} \vdash_\Sigma \alpha\}$. By definition, $\mathrm{Ax} \subseteq \mathrm{Ax}^* \subseteq Sen_{\mathrm{Object}}(\Sigma)$.

Definition B.17. A *theory* is a specification in which the axioms are closed under entailment. Specification <Σ, Ax> is a *presentation* of the theory <Σ, Ax^*>.

There may be many different presentations for a given theory.

As an aside, a model-based approach, in contrast to the algebraic (property-based) approach, does not have logic-based axioms. Instead of logic, a reference model is used like sets, sequences or relations and the axioms refer directly to those concepts and describe operations in terms of their explicit effect on state (often using pre and post conditions).

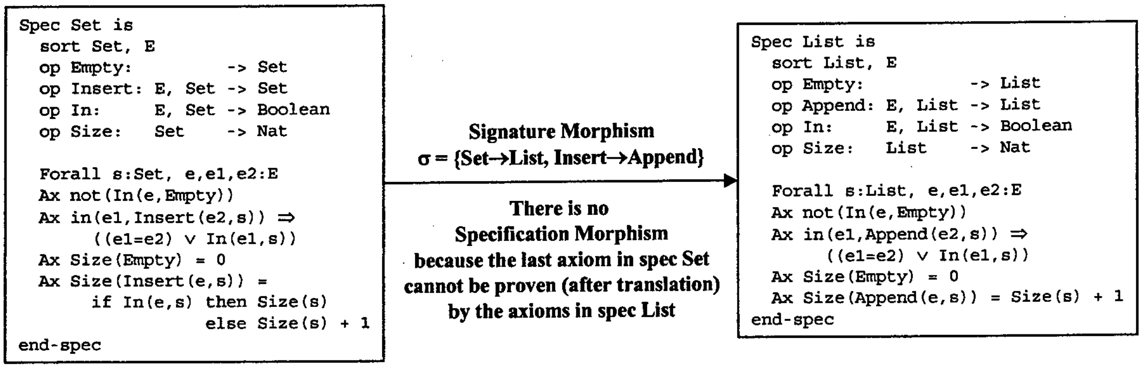
B.1.7 Specification morphisms

A specification morphism is a relationship between source and target specifications in which the target specification has at least as much structure (interface and properties) as the source. A specification morphism is a signature morphism between the signatures of the specifications where all the (translated) axioms of the source specification are theorems of the target specification. This mapping insures that the interface (signature) of an abstract data type is preserved and the properties (axioms) over that interface are preserved. Thus if the source specification represents the requirements for a system and the target represents the implementation, then the existence of a specification morphism between them is proof that the implementations meets all the specified requirements.

Definition B.18. A *specification morphism* σ: A → B for A = <Σ, Ax> and B = <Σ', Ax'> is a signature morphism f: Σ → Σ' in which for all α ∈ Ax, Ax' $\vdash_{\Sigma'} f(\alpha)$

where $f = Sen_{\mathrm{Arrow}}(f)$ is a function that translates Σ-sentences to Σ'-sentences.

In Figure B-2, there is a signature morphism, σ, from specification Set to specification List indicated by the pair of relations <{Set→List}, {Insert→Append}>. The first three (translated) axioms of specification Set are obviously theorems of specification List. However, the last axiom of specification Set is not a theorem of specification List (in any reasonable entailment system) and therefore the signature morphism is not a specification morphism. Since there are no other signature morphisms from Set to List there are no specification morphisms from Set to List.



```
Spec Set is
  sort Set, E
  op Empty:         -> Set
  op Insert: E, Set -> Set
  op In:     E, Set -> Boolean
  op Size:   Set    -> Nat

  Forall s:Set, e,e1,e2:E
  Ax not(In(e,Empty))
  Ax in(e1,Insert(e2,s)) ⇒
       ((e1=e2) v In(e1,s))
  Ax Size(Empty) = 0
  Ax Size(Insert(e,s)) =
       if In(e,s) then Size(s)
                  else Size(s) + 1
end-spec
```

Signature Morphism
σ = {Set→List, Insert→Append}

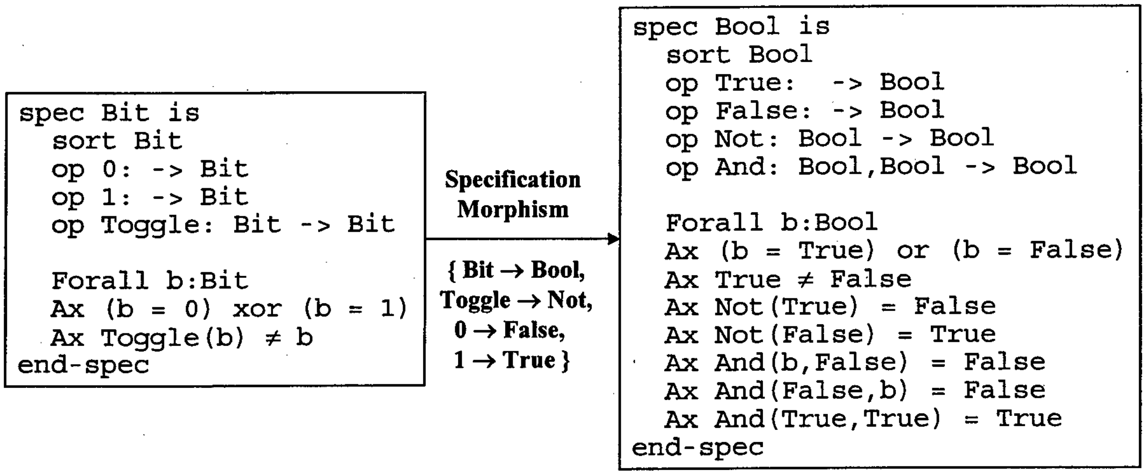
There is no
Specification Morphism
because the last axiom in spec Set
cannot be proven (after translation)
by the axioms in spec List

```
Spec List is
  sort List, E
  op Empty:          -> List
  op Append: E, List -> List
  op In:     E, List -> Boolean
  op Size:   List    -> Nat

  Forall s:List, e,e1,e2:E
  Ax not(In(e,Empty))
  Ax in(e1,Append(e2,s)) ⇒
       ((e1=e2) v In(e1,s))
  Ax Size(Empty) = 0
  Ax Size(Append(e,s)) = Size(s) + 1
end-spec
```

Figure B-2. Signature morphism Set → List

Although there is also an inverse Σ-morphism from specification List to specification Set there are no specification morphisms in that direction either because the last List axiom is not a theorem (after translation) of the Set specification. In practical terms, an implementation of the List specification can not be used to implement the Set specification (and vice versa) without being extended somehow. The last axioms in both specifications indicate the set operations *In*, *Insert* and *Size* interact in a different way than the list operations *In*, *Append* and *Size* interact and thus the interface of one cannot be directly used to implement the other. As shall be illustrated in Appendix C, one must often extend a specification in order for it to be a suitable target for a specification morphism.

In Figure B-3 the signature morphism from Bit to Bool, {Bit→Bool, Toggle→Not, 0→False, 1→True}, is also a specification morphism as the axioms of specification Bit are theorems of specification Bool. Interestingly enough, the signature morphism that assigns the constant functions 0 and 1 to True and False respectively is also a specification morphism leading to the realization that assigning 0 to false and 1 to true are arbitrary choices.



```
spec Bit is
  sort Bit
  op 0: -> Bit
  op 1: -> Bit
  op Toggle: Bit -> Bit

  Forall b:Bit
  Ax (b = 0) xor (b = 1)
  Ax Toggle(b) ≠ b
end-spec
```

```
spec Bool is
  sort Bool
  op True:  -> Bool
  op False: -> Bool
  op Not: Bool -> Bool
  op And: Bool,Bool -> Bool

  Forall b:Bool
  Ax (b = True) or (b = False)
  Ax True ≠ False
  Ax Not(True) = False
  Ax Not(False) = True
  Ax And(b,False) = False
  Ax And(False,b) = False
  Ax And(True,True) = True
end-spec
```

Figure B-3. Specification morphism: Bit → Bool

There is no signature or specification morphism from specification Bool to specification Bit although one could (manually) extend specification Bit with additional operations to make such a signature and specification morphism possible.

B.1.8 The category *Spec*

Proposition B.19. Specifications and specification morphisms form a category, *Spec*, where the objects are specifications and the arrows are specification morphisms.

Proof: We already know that the signature portion of a specification forms a category *Sign*. Therefore we concentrate on the axioms and assume that for every specification A there is a *Sign*-object **A** and for every specification morphism f: A→B there is a *Sign*-arrow $\boldsymbol{f}$: **A→B**.

Identity Arrows: The identity arrow for each specification A, id_A, is the identity signature morphism, $\mathbf{id}_A$, for **A** as $Sen_{Arrow}(\mathbf{id}_A)$ translates **A**-sentences to **A**-sentences (which is essentially a no-op) and each axiom of A is trivially a theorem of A by the reflexivity of $\vdash$, i.e. $Sen_{\text{Arrow}}(\text{id}_A) = \text{id}(Sen_{\text{Object}}(A))$.

Arrow Composition: For specification morphisms f: A→B and g:B→C there is always a specification morphism $g \circ f$: A→C as the underlying signature morphisms compose (i.e. $\boldsymbol{g \circ f}$**: A→C**) and the entailment relation is transitive across specification morphisms, i.e.

$\text{axioms(B)} \vdash_{\text{Signature(B)}} f_t(\text{axioms(A)})$	❶ Given
$\text{axioms(C)} \vdash_{\text{Signature(C)}} g_t(\text{axioms(B)})$	❷ Given
$g_t\,(\text{axioms(B)}) \vdash_{\text{Signature(C)}} g_t(\,f_t(\text{axioms(A)}))$	$\vdash$-translation and ❶
$\text{axioms(C)} \vdash_{\text{Signature(C)}} g_t(\,f_t(\text{axioms(A)}))$	monotonicity of $\vdash$, transitivity of $\vdash$, and ❷
$\text{axioms(C)} \vdash_{\text{Signature(C)}} (\,g_t \circ f_t)(\text{axioms(A)})$	composition of *Set*-arrows

Where the functions f_t and g_t translate **A**-sentences to **B**-sentences and **B**-sentences to **C**-sentences respectively, i.e. $f_t = Sen_{\text{Arrow}}(\boldsymbol{f})$ and $g_t = Sen_{\text{Arrow}}(\mathbf{g})$.

Arrow Associativity: Given specification morphisms f: A→B, g:B→C, and h:C→D then $h \circ (g \circ f) = (h \circ g) \circ f$ as the underlying signature morphisms $\boldsymbol{h \circ (g \circ f) = (h \circ g) \circ f}$ are associative and we have just proven that specification morphisms compose. □

In several places above we have discussed the signature of a specification or the signature morphism between two specifications. This relationship between specification and signature is captured by the functor between the *Spec* and *Sign* categories. The functor *Sig*:*Spec*→*Sign* is a forgetful functor that maps *Spec* objects to *Sign* objects by dropping the axioms from a specification and maps *Spec* arrows to *Sign* arrows using the underlying signature morphism. Forgetful functors make it possible to expose the structure of objects and arrows in a category by relating a structured category to a simpler one by forgetting some of the structure.

As categories and functors are the objects and arrows of a category *Cat* [AHS90], functors can be composed. We define *Sen*:*Spec*→*Set* to be *Sen*:*Sign*→*Set* ∘ *Sig*:*Spec*→*Sign* as depicted in Figure B-4. Note the overloading of the name *Sen* (i.e. *Sen*:*Spec*→*Set* and *Sen*:*Sign*→*Set*).

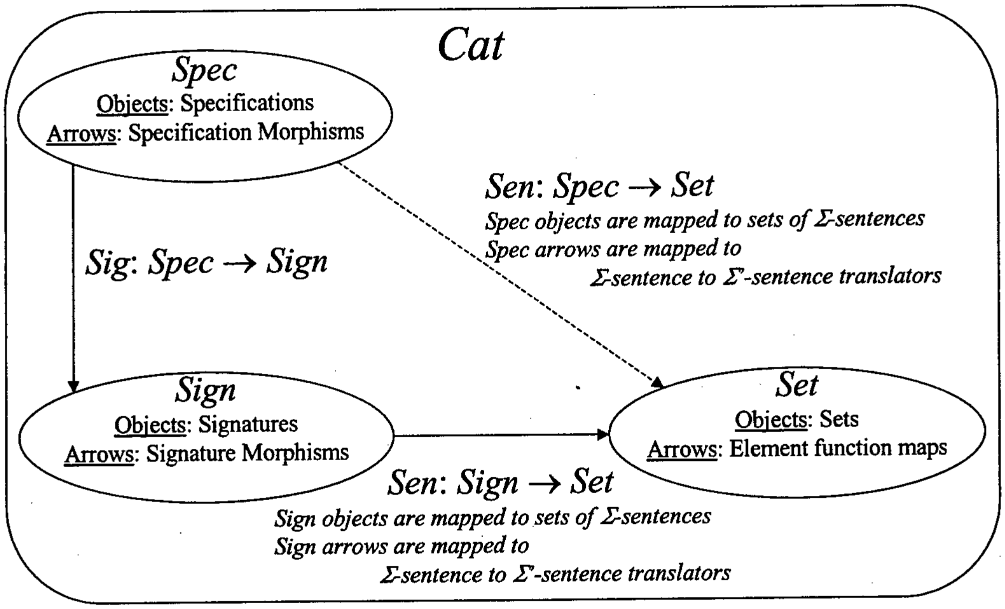

Cat
Spec
Objects: Specifications
Arrows: Specification Morphisms
Sen: Spec → Set
Spec objects are mapped to sets of Σ-sentences
Spec arrows are mapped to
Σ-sentence to Σ'-sentence translators
Sig: Spec → Sign
Sign
Objects: Signatures
Arrows: Signature Morphisms
Set
Objects: Sets
Arrows: Element function maps
Sen: Sign → Set
Sign objects are mapped to sets of Σ-sentences
Sign arrows are mapped to
Σ-sentence to Σ'-sentence translators

Figure B-4. Algebraic categories and the *Sen* Functors

An isomorphism in *Set* is based on the number of elements in the domain and codomain *Set*-objects being the same. An isomorphism in *Sign* is based on the structure of the signature (sorts, operations, and operation ranks) in the domain and codomain *Sign*-objects being the same. For *Spec*, an isomorphism is based on the signature as well as what can be proved from the axioms. In *Spec*, if there is a signature isomorphism between two specifications and the axioms of one can be used to prove the axioms of the other (and vice-versa) then there is an isomorphism between them. Another way of stating this is that two specifications are isomorphic if their (translated) theories are the same.

Category theory enables us to consider two specifications with isomorphic signatures but different axioms to be isomorphic when the entailment relation enables the axioms to prove the same theory. This is one method by which we can add design information to a specification without changing its meaning (theory). One set of axioms can abstractly define an input/output relation for a function. A different set of axioms can concretely define the same input/output relation "algorithmically". Both sets of axioms can generate the same theory. Because the entailment relation is monotonic and transitive we do not have to generate and relate both theories, merely prove that each set of axioms entails the other.

B.2 Models

In this section the relationship between algebraic specifications and models is defined. An algebraic specification can be viewed as a formal requirement document as the models that satisfy the requirement specification are the possible (abstract) implementations of those requirements. When specifying requirements it is important that the models associated with a specification include all desired implementations and exclude those that are not desired. In general this may be difficult, as determining if a model is desired is at heart a validation problem instead of a verification problem. A design decision adds additional properties to the specification and may reduce the class of models associated with the specification. An objective of refinement (and design) is to reduce the class of models by imposing additional structure and constraints on the implementation while preserving all of the original properties.

A concrete implementation such as a set of FORTRAN or Ada subprograms over intrinsic or constructed data types is also a model. Such concrete implementations are the ultimate goal of refinement but are too restrictive to work with and reason about as requirements because of their syntactic baggage and operational semantics.

In this section specification models are defined by first defining *signature models* and *signature model homomorphisms*, which are the models associated with a signature and the arrows between those models. Signature model categories and the *reduct functors* between them are defined in a way that is parallel to the signatures and signature morphisms from which they are derived. Finally *satisfaction* is defined as the relationship between a signature model and a signature sentence and a specification model is defined as a signature model that satisfies all of the axioms of a specification. Each specification has associated with it a restriction of its signature model category and each specification morphism has associated with it a restriction of the reduct functor between the signature model categories.

B.2.1 Signature models

Definition B.20. A *signature model*, also called an *Σ-model*, consists of a sort-indexed collection of carrier sets and functions over those sets, one for each sort and operation in Σ, that are rank consistent. Thus if $\Sigma = <S, \Omega>$ then a Σ-model $= <A_S, F_A>$ where $A_S = \{A_s \mid s \in S\}$ is the set of carrier sets indexed by $s \in S$ and $F_A = \{f_A \mid f \in \Omega\}$ is the set of functions associated with each $f \in \Omega$ such that if the rank of f is $s_1, s_2, \ldots,$

$s_n \rightarrow s$ then the function f_A is from $A_{s_1} \times A_{s_2} \times \ldots \times A_{s_n}$ to A_s, where × is the Cartesian product of sets. The collection of Σ-models for a given Σ is denoted by Mod(Σ).

The definition for signature models does not constrain the carrier sets in terms of the number or type of elements, nor does it constrain the functions over those sets in terms of their functionality. Because of this lack of constraint, most signature models are not what is wanted as a possible implementation of a specification.

As an example, the signature BOOL = <{Bool}, {True:→Bool, False:→Bool, Not:Bool→Bool}> has one sort and three operations, two of which are constants. There are an infinite number of models in Mod(BOOL) besides the standard one.

Example B.21. In the so-called standard model for signature BOOL, the carrier set for sort Bool has two elements. The operators associated with True and False return different Bool carrier set elements. The value of the function Not returns the element not supplied as a parameter.

Model $\text{BOOL}_{\text{Standard}} = \text{Bool} \overset{\text{def}}{=} \{0,1\}$ where $\text{True} \overset{\text{def}}{=} 1$, $\text{False} \overset{\text{def}}{=} 0$, $\text{Not}(1) \overset{\text{def}}{=} 0$, $\text{Not}(0) \overset{\text{def}}{=} 1$

Example B.22. In the so-called final model for signature BOOL, the carrier set for sort Bool has one element (value). Both function constants return this element as does the function Not.

Model $\text{BOOL}_{\text{Final}} = \text{Bool} \overset{\text{def}}{=} \{1\}$ where $\text{True} \overset{\text{def}}{=} 1$, $\text{False} \overset{\text{def}}{=} 1$, $\text{Not}(1) \overset{\text{def}}{=} 1$

Example B.23. In a so-called loose model for signature BOOL, the carrier set for sort Bool has three elements. The constant functions True and False return different Bool elements. The Not operation "swaps" the bool carrier set values of 0 and 1 and merely echoes the value of ⊥.

Model $\text{BOOL}_{\text{Loose}} = \text{Bool} \overset{\text{def}}{=} \{0,1,\perp\}$where $\text{True} \overset{\text{def}}{=} 1$, $\text{False} \overset{\text{def}}{=} 0$, $\text{Not}(1) \overset{\text{def}}{=} 0$, $\text{Not}(0) \overset{\text{def}}{=} 1$, $\text{Not}(\perp) \overset{\text{def}}{=} \perp$

Example B.24. In a different so-called loose model for signature BOOL, the carrier set for sort Bool has four elements. True and False return different Bool elements and the function Not has "odd" functionality as shown below.

Model $\text{BOOL}_{\text{Odd}} = \text{Bool} \overset{\text{def}}{=} \{0,1,\alpha,\beta\}$where $\text{True} \overset{\text{def}}{=} 1$, $\text{False} \overset{\text{def}}{=} 0$, $\text{Not}(1) \overset{\text{def}}{=} 1$, $\text{Not}(0) \overset{\text{def}}{=} \alpha$, $\text{Not}(\alpha) \overset{\text{def}}{=} 0$, $\text{Not}(\beta) = \beta$

All four example BOOL-models are valid implementations of the signature BOOL. The labels initial, final and loose are based on the terms that can be generated for a given signature. As can be extrapolated from these examples, even simple signatures such as BOOL have an infinite number of models. Only some of these models are interesting and useful.

All models in a given Mod(Σ) have carrier sets and functions that relate to the same signature but have differing elements in their carrier sets and differing functions over those carrier sets. It makes sense to relate the Σ-models in Mod(Σ) based on these differing features. A Σ-homomorphism is a mapping between the elements of the signature-related carrier sets of two Σ-models in a way that is compatible with all of the signature-related functions.

B.2.2 Signature model homomorphisms

Definition B.25. Given a signature Σ = <S, Ω> and two Σ-models A = <A_S, F_A> and B = <B_S, F_B>, a *signature homomorphism*, also called a *Σ-homomorphism*, h:A→B is a family of sort-indexed functions $\{h_s:A_s \to B_s \mid s \in S\}$ between the carrier sets (mapping their elements) such that the operations over the carrier set elements are compatible. More formally, for all Ω operations f: $s_1, s_2, \ldots s_n \to s$, and the associated A and B Σ-algebra functions f_A: $A_{s_1} \times A_{s_2} \times \ldots \times A_{s_n}$ to A_s and f_B: $B_{s_1} \times B_{s_2} \times \ldots \times B_{s_n}$ to B_s, and for all carrier set elements $a_1 \in A_{s_1}$ $a_2 \in A_{s_2}$ $\ldots a_n \in A_{s_n}$

$$h_s(f_A(a_1, a_2, \ldots, a_n)) = f_B(h_{s_1}(a_1), h_{s_2}(a_2), \ldots, h_{s_n}(a_n))$$

The four example models for the signature BOOL given above have several Σ-homomorphisms between them.

Example B.26. The Σ-homomorphism between $\text{BOOL}_{\text{Standard}}$ and $\text{BOOL}_{\text{Final}}$ is

$$h = \{ h_{\text{Bool}} \mid h_{\text{Bool}}(0_{\text{Standard}}) = 1_{\text{Final}} \land h_{\text{Bool}}(1_{\text{Standard}}) = 1_{\text{Final}} \}$$

as for all b ∈ Bool; $h_{\text{Bool}}(\text{True}_{\text{Standard}}) = \text{True}_{\text{Final}}$, $h_{\text{Bool}}(\text{False}_{\text{Standard}}) = \text{False}_{\text{Final}}$, and $h_{\text{Bool}}(\text{Not}_{\text{Standard}}(b)) = \text{Not}_{\text{Final}}(h_{\text{Bool}}(b))$.

Example B.27. The Σ-homomorphism between $\text{BOOL}_{\text{Standard}}$ and $\text{BOOL}_{\text{Loose}}$ is

$$h = \{ h_{\text{Bool}} \mid h_{\text{Bool}}(0_{\text{Standard}}) = 0_{\text{Loose}} \land h_{\text{Bool}}(1_{\text{Standard}}) = 1_{\text{Loose}} \}$$

as for all b ∈ Bool, $h_{\text{Bool}}(\text{True}_{\text{Standard}}) = \text{True}_{\text{Loose}}$, etc.

Example B.28. The Σ-homomorphism between $\mathrm{BOOL}_{\mathrm{Odd}}$ and $\mathrm{BOOL}_{\mathrm{Final}}$ is

$$h = \{\, h_{\mathrm{Bool}} \mid h_{\mathrm{Bool}}(0_{\mathrm{Odd}}) = 1_{\mathrm{Final}} \wedge h_{\mathrm{Bool}}(1_{\mathrm{Odd}}) = 1_{\mathrm{Final}} \wedge h_{\mathrm{Bool}}(\alpha_{\mathrm{Odd}}) = 1_{\mathrm{Final}} \wedge h_{\mathrm{Bool}}(\beta_{\mathrm{Odd}}) = 1_{\mathrm{Final}} \,\}$$

as for all b ∈ Bool, $h_{\mathrm{Bool}}(\mathrm{True}_{\mathrm{Odd}}) = \mathrm{True}_{\mathrm{Final}}$, etc.

There is no Σ-homomorphism $h{:}\mathrm{BOOL}_{\mathrm{Loose}} \rightarrow \mathrm{BOOL}_{\mathrm{Standard}}$ as the $\mathrm{Bool}_{\mathrm{Loose}}$ element ⊥ cannot be mapped to either $\mathrm{Bool}_{\mathrm{Standard}}$ element so that the equation $h(\mathrm{Not}_{\mathrm{Loose}}(\perp)) = \mathrm{Not}_{\mathrm{Standard}}\,(h(\perp))$ is true. Note that the carrier set elements are defined in and by the model. The choice of 0, 1 and ⊥ as the symbols of the carrier set elements are "syntactic conveniences" and do not convey a deeper meaning in and of themselves.

B.2.3 Signature model categories

The collection of Σ-models and the Σ-homomorphisms between them are a category of Σ-models. The initial model, if one exists, has an arrow from it to all other models. The final model, if one exists, has an arrow to it from all other models.

Proposition B.29. Σ-models and Σ-homomorphisms form a category, *Mod(Σ)*, where the objects are Σ-models and the arrows are Σ-homomorphisms.

Proof: Assume that for a signature Σ = <S, Ω> there exist arbitrary Σ-models A = $<A_S, F_A>$, B = $<A_S, F_B>$, C = $<C_S, F_C>$, and D = $<D_S, F_D>$ with Σ-homomorphisms $h{:}A \rightarrow B$, $j{:}B \rightarrow C$, and $k{:}C \rightarrow D$.

Identity Arrows: The identity Σ-homomorphism is the collection of identity functions for each carrier set of Sorts(Σ) associated with each Σ-model object. These identity arrows map the carrier set elements to themselves, i.e. the identity Σ-homomorphism $\mathrm{id}_A = \{h_s \mid s \in S \text{ where for all } \alpha \in A_s,\ h_s(\alpha) = \alpha)\}$

Arrow Composition: for the f_A, f_B, and f_C functions associated with all $f \in \Omega$ and all elements $a_1, a_2, \ldots a_n$ and $b_1, b_2, \ldots b_n$ associated with each of the carrier sets $S_1, S_2, \ldots S_n \in S$ in models A and B

$$h_s(f_A(a_1,a_2,\ldots,a_n)) = f_B(h_{s_1}(a_1), h_{s_2}(a_2),\ldots,h_{s_n}(a_n)) \qquad \underline{\text{Given}}$$

$$j_s(f_B(b_1,b_2,\ldots,b_n)) = f_C(j_{s_1}(b_1), j_{s_2}(b_2),\ldots,j_{s_n}(b_n)) \qquad \underline{\text{Given}}$$

$$j_s(f_B(h_{s_1}(a_1),h_{s_2}(a_2),\ldots,h_{s_n}(a_n))) = f_C(j_{s_1}(h_{s_1}(a_1)), j_{s_2}(h_{s_2}(a_2)),\ldots,j_{s_n}(h_{s_n}(a_n))) \qquad \underline{\text{Substitution}}$$

$$j_s(h_s(f_A(a_1,a_2,\ldots,a_n))) = f_C(j_{s_1}(h_{s_1}(a_1)), j_{s_2}(h_{s_2}(a_2)),\ldots,j_{s_n}(h_{s_n}(a_n))) \qquad \underline{\text{Substitution}}$$

Σ-homomorphisms compose as functions over sets compose.

Arrow Associativity: Through reasoning similar to that for arrow composition, and because functions over sets are associative, Σ-homomorphisms are associative. □

Isomorphisms in the category *Mod(Σ)* are "translations" of carrier set elements and functions. For example the model $\text{BOOL}_{\text{Standard-2}} = \text{Bool} \overset{\text{def}}{=} \{\mu, \nu\}$ where True $\overset{\text{def}}{=} \mu$, False $\overset{\text{def}}{=} \nu$, $\text{Not}(\mu) \overset{\text{def}}{=} \nu$, $\text{Not}(\nu) \overset{\text{def}}{=} \mu$ is isomorphic to the model $\text{BOOL}_{\text{Standard}}$ in Example B.21. A monomorphism is a one to one mapping of carrier set elements. An epimorphism has every target carrier set element mapped to by at least one source carrier set element.

Mod(Σ) has as objects all models of a given signature Σ. However, as we shall see when we get to models of specifications only a subset of those models are of interest.

B.2.4 Reduct functors

If there is a signature morphism, *f*: A→B, how are the Σ-models of A and B related? There is a mapping from the sorts and operations of signature A to the carrier sets and functions of each B-model because of the signature morphism *f*: A→B. If unmapped carrier sets and functions are removed from each B-model, then what is left must be an A-model.

Definition B.30. Given a signature A = <S, Ω>, a signature morphism σ:A→B and a B-model $\beta = <B_S, F_B>$, the (object) *σ-reduct* of β, denoted $\beta|_\sigma$ is the A-model $\beta|_\sigma = <A_S, F_A>$ where $A_S = \{B_{\sigma(s)} \mid s \in S\}$ and $F_A = \{ \sigma(f)_B \mid f \in \Omega\}$. The carrier sets and functions in the model not associated with signature A sorts and operations are removed (forgotten).

Definition B.31. Given a signature A = <S, Ω>, signature morphism σ:A→B and a *Mod(B)* arrow $h{:}\beta_1 \rightarrow \beta_2$ the (arrow) *σ-reduct* of *h*, denoted $h|_\sigma$ is the *Mod(A)* arrow $h|_\sigma{:}\beta_1|_\sigma \rightarrow \beta_2|_\sigma$ where the $h|_\sigma$ Σ-homomorphism is the family of functions in *h* restricted to the sorts in the signature of A, i.e. $h|_\sigma = \{h_{\sigma(s)} \mid s \in \mathit{Sorts}(A)\}$, and $\beta_1|_\sigma$ and $\beta_2|_\sigma$ are the (object) σ-reducts $\beta_1$and β_2.

For every signature A, there is a category of Σ-models, *Mod(A)*. For a signature morphism σ:A→B, how are the model categories *Mod(A)* and *Mod(B)* related? There is a functor that maps each

Mod(B)-object to its σ-reduct object in *Mod(A)* and each *Mod(B)*-arrow to its σ-reduct arrow in *Mod(A)*, see Figure B-5.

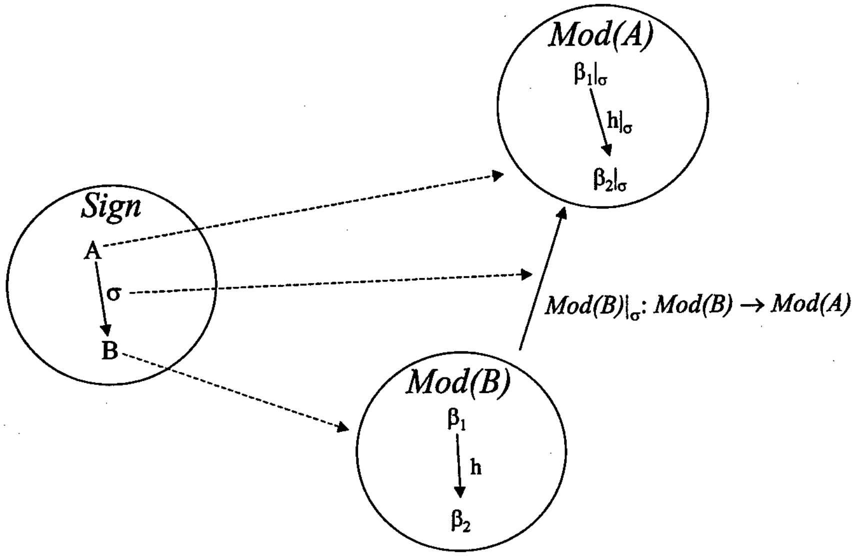


Figure B-5. Each Σ and Σ-morphism in *Sign* has an associated model category and functor

Proposition B.32. Given a signature morphism $\sigma{:}A \rightarrow B$, the (functor) *σ-reduct* from *Mod(B)* to *Mod(A)*, $Mod(B)|_\sigma : Mod(B) \rightarrow Mod(A)$, maps each *Mod(B)*-object β to its σ-reduct object, $\beta|_\sigma$, in *Mod(A)* and each *Mod(B)*-arrow h: $\beta_1 \rightarrow \beta_2$ to the *Mod(A)*-arrow $h|_\sigma{:}\beta_1|_\sigma \rightarrow \beta_2|_\sigma$.

Proof:

Preservation of Identity Arrows: Let id_β be the identity arrow for $\beta \in$ Objects*(Mod(B))*. The identity arrow of the *Mod(A)* object $\beta|_\sigma$ requires a family h of sort-indexed functions $\{h_s{:}A_s \rightarrow B_s \mid s \in S\}$ for each of its carrier sets that maps the elements to themselves. The *Mod(B)*-arrow id_β has that property for the carrier sets associated with the sorts in *Sorts*(B). The *Mod(A)*-arrow($id_{\beta|\sigma}$) does as well because *Sorts*(A) ⊆ *Sorts*(B) and σ-reduct$_{\text{Arrow}}$(id_β) function restricts the carrier set identity functions of id_β to those over the carrier sets associated with the sorts in *Sorts*(A), i.e. *σ-reduct*$_{\text{Arrow}}$(id_β) = $id_\beta|_\sigma$ = $id_{\sigma\text{-}reduct_{\text{Object}}(\beta)}$.

Preservation of Arrow Composition: Let $f{:}\beta_1 \rightarrow \beta_2$ and $g{:}\beta_2 \rightarrow \beta_3$ be elements of Arrows*(Mod(B))* where f and g are Σ-homomorphisms. (Sets of functions $\{f_s \mid s \in S\}$ and $\{g_s \mid s \in S\}$ that map carrier set elements in a way that is compatible with the model functions.) The function *σ-reduct*$_{\text{Arrow}}$($g \circ f$) takes a *Mod(B)* Σ-homomorphism ($g \circ f$): $\beta_1 \rightarrow \beta_3$ to a *Mod(A)* Σ-homomorphism $(g \circ f)|_\sigma{:}\beta_1|_\sigma \rightarrow \beta_3|_\sigma$ by first composing the indexed carrier set functions of f and g and then restricting the composite set of functions based on the (arrow) σ-reduct. The composition $\sigma\text{-}reduct_{\text{Arrow}}(g){:}\beta_2|_\sigma \to \beta_3|_\sigma \circ \sigma\text{-}reduct_{\text{Arrow}}(f){:}\beta_1|_\sigma \to \beta_2|_\sigma$ restricts the functions based on the (arrow) σ-reduct first and then composes the functions, which results in the same *Mod(A)* arrow, i.e. $\sigma\text{-}reduct_{\text{Arrow}}(g \circ f) = \sigma\text{-}reduct_{\text{Arrow}}(g) \circ \sigma\text{-}reduct_{\text{Arrow}}(f)$. □

For every signature Σ there may be many Σ-models, most of which are not desired. For example, since signatures SET and LIST (from Example B.2 and Example B.3) are isomorphic objects in *Sign*, Mod(SET) and Mod(LIST) are the same (isomorphic) collection of Σ-models. Obviously only some of the Σ-models fit the notion of a "set" and others the notion of a "list". Selecting only those Σ-models that satisfy certain Σ-sentences is one way to eliminate unwanted algebras.

B.2.5 Specification models

Specifications contain axioms that are logical sentences over the signature. Those logical sentences must be true of a model of the specification; therefore there needs to be a relationship between Σ-models and Σ-sentences. The relationship must be sound in that if the axioms of a specification entail additional Σ-sentences then so will all models of that specification.

Definition B.33. *Satisfaction* is a relation between a Σ-model and a Σ-sentence, written $M \models_\Sigma \alpha$, that is used when Σ-sentence α is true for model M. The relationship between entailment, $\vdash_\Sigma$, and satisfaction, $\models_\Sigma$, is defined as follows: if $M \models_\Sigma \alpha$ for all $\alpha \in \Gamma$ and $\Gamma \vdash_\Sigma \beta$ then $M \models_\Sigma \beta$.

The satisfaction relation is used to identify the Σ-models that satisfy the properties described by the axioms of a given specification over signature Σ.

Definition B.34. A *specification model* of specification <Σ, Ax> is a signature model, $M \in \text{Mod}(\Sigma)$, where M satisfies all axioms, Ax, of the specification, i.e. $M \models_\Sigma \alpha$ for all $\alpha \in \text{Ax}$.

The collection of models associated with a specification A is denoted Mod(A). By definition, the collections of all models of a specification A is a subset of the models of the signature of A, $\text{Mod(A)} \subseteq \text{Mod}(\mathit{Signature}(\text{A}))$.

Proposition B.35. If $M \in \text{Mod(A)}$ and α is a theorem of specification A then $M \models_{\text{Signature(A)}} \alpha$.

Proof: Immediate from Definition B.33 and Definition B.34.

Example B.36. Given the specification Bool in Figure B-6, Mod(Bool) is the set of all Bool signature models that are isomorphic to $\text{BOOL}_{\text{Standard}}$ (Example B.21), i.e. Mod(Bool) = $\{\text{std} \mid \text{std} \cong \text{BOOL}_{\text{Standard}}\}$. The first axiom in specification Bool, "`True ≠ False`", eliminates the $\text{BOOL}_{\text{Final}}$ model (Example B.22) as well as any other models that have the same carrier set values for the constant functions True and False. The last axiom, "`b = True OR b = False`", eliminates all models where there are more than two elements in the Bool carrier set. The middle two axioms fully define the functionality of the *Not* function and eliminate any models that don't have that functionality. Thus the axioms constrain models of the specification Bool to have two elements in the single carrier set Bool, with True ≠ False and the function Not defined in the usual way. The theorem "`Not(Not(b)) = b`" can be proven from the last three axioms and therefore by Proposition B.35, all specification models will satisfy that theorem because they satisfy the last three axioms.

```
Spec Bool is
  sort Bool
  op True:  -> Bool
  op False: -> Bool
  op Not: Bool -> Bool

  Forall b:Bool
  Ax True ≠ False
  Ax Not(True) = False
  Ax Not(False) = True
  Ax b = True OR b = False

  Theorem Not(Not(b)) = b
end-spec
```

Figure B-6. Simple specification for Bool

Axioms characterize the sorts and operations of a specification by relating terms of the specification. These axioms place two types of constraints on the models associated with a specification: they constrain the elements of the carrier sets and they constrain the input/output relation of the functions.

There are two ways that axioms place restrictions on carrier set elements. One way enumerates them to specific terms such as axiom "`∀ b:Bool | b = True or b = false`." The Bool-terms for which there are (possibly) distinct carrier set elements are individually listed. Another method is to equate or "dis"-equate terms. The axiom "`True ≠ False`" means that the carrier set elements that are returned by constant functions True and False are not equal to each other. It says nothing about how many other carrier set elements there may be. Together, however, the two axioms limit the carrier set of Bool to exactly two elements. Either axiom by itself is not sufficient to characterize the carrier set.

A term can be thought of as a "handle" for a carrier set element. If no term exists for a carrier set element, then there is no way to reference that carrier set element. Listing the terms is practical for sorts like Bool whose (standard) carrier sets have a small finite number of elements but it is not practical in general. Instead, an induction scheme is used to describe the terms that are carrier set elements. The axiom "`∀ s:Set ∃ t:Set,e:E | s = Empty OR s = insert(e,t)`" implies that every carrier set element for sort Set must be the constant Empty or must be derivable from Empty by inserting E's into Empty. The text "`Constructors {Empty, Insert} construct Set`" in Figure B-7 is syntactic sugar for a similar induction axiom. The last two axioms in Figure B-6 can be viewed as using equality between terms to assert that some of the terms generated by the induction scheme are result in the same carrier set elements.

```
spec Set is
  sort Set, E
  op Empty:          -> Set
  op Insert: E, Set -> Set
  op In:     E, Set -> Boolean

  Constructors {Empty, Insert} construct Set

  Forall s:Set, e,e1,e2:E
  Ax not(In(e,Empty))
  Ax in(e1,Insert(e2,s)) ⇒ ((e1=e2) ∨ In(e1,s))
  Ax Insert(e,s) = Insert(e,Insert(e,s))
  Ax Insert(e1,Insert(e2,s)) = Insert(e2,Insert(e1,s))
end-spec
```

Figure B-7. Constructor axiom for sort Set

B.2.6 Specification model homomorphisms

A *homomorphism* between models of a specification SP in Mod(SP) is defined in the same manner as a Σ-homomorphism between Σ-models of a signature Σ is defined in Definition B.25.

Definition B.37. Given a specification SP = <S, Ω, Ax> and two spec-models of SP, A = $<A_S, F_A>$ and B = $<B_S, F_B>$, a *homomorphism*, *h*: A→B is defined to be the Σ-homomorphism between the two models.

Proposition B.38. Models of a specification A, Mod(A), and the Σ-homomorphisms between them are the objects and arrows in a full subcategory, *Mod(A)*, of the category *Mod(Signature(A))*.

Proof: Objects*(Mod(A))* $\subseteq$ Objects*(Mod(Signature(A)))* by definition. The Σ-homomorphisms between *Mod(A)* objects are the full set of arrows between those same objects in *Mod(Signature(A))* by definition. Therefore *Mod(A)* is a full subcategory of *Mod(Signature(A))*. □

Specification model categories have reduct functors between them based on the specification morphisms between them, as depicted in Figure B-8. The ρ-reduct functor over specification model categories is a restriction in both the domain and codomain categories of the σ-reduct functor between signature model categories.

Proposition B.39. For any specification morphism ρ:A→B and associated signature morphism σ:Signature(A)→Signature(B) there is a specification model category reduct functor, ρ-reduct: *Mod(B)→Mod(A)*, which is a restriction of the signature model category reduct functor, σ-reduct: *Mod(Signature(B))→Mod(Signature(A))* where $\rho\text{-reduct}_{\text{Object}}(\beta) = \sigma\text{-reduct}_{\text{Object}}(\beta)$ for all $\beta \in$ Objects*(Mod(B))* and $\rho\text{-reduct}_{\text{Arrow}}(h) = \sigma\text{-reduct}_{\text{Arrow}}(h)$ for all $h \in$ Arrows*(Mod(B))*.

Proof:

Domain Restriction: *Mod(B)* is a full subcategory of *Mod(Signature(B))*. Thus the signature model category reduct functor, σ-reduct: *Mod(Signature(B))→Mod(Signature(A))*, can be restricted in its domain to operate only over the objects and arrows in *Mod(B)*, i.e. *Mod(B)→Mod(Signature(A))*. One can also view this as the composition of the inclusion function from *Mod(B)→Mod(Signature(B))* with the signature reduct functor, σ-reduct: *Mod(Signature(B))→Mod(Signature(A))*.

Codomain restriction: *Mod(A)* is a full subcategory of *Mod(Signature(A))*. By the domain restriction above we know that for all models M $\in$ Objects*(Mod(B))* the σ-reduct of M, $M|_\sigma$, is in *Mod(Signature(A))*. Is the $M|_\sigma$ also in *Mod(A)*, i.e. does $M|_\sigma \models_{\text{Signature(A)}} \alpha$ for all $\alpha \in$ Axioms(A)? We know that for all models M $\in$ Objects*(Mod(B))*, $M \models_{\text{Signature(B)}} \rho(\alpha)$ for all $\alpha \in$ Axioms(A) where ρ is an axiom translation function as Axioms(B) $\vdash_{\text{Signature(B)}} \rho(\alpha)$ and Definition B.33 ensures that theorems of a specification are satisfied by models of a specification. A model $M|_\sigma$ may have fewer carrier sets and functions than model M; however, the additional carrier sets and functions in M do not change the properties of the existing carrier sets and functions. Since the axioms of A are over the reduced signature of $M|_\sigma$, $M|_\sigma \models_{\text{Signature(A)}} \alpha$ for all $\alpha \in$ Axioms(A). □

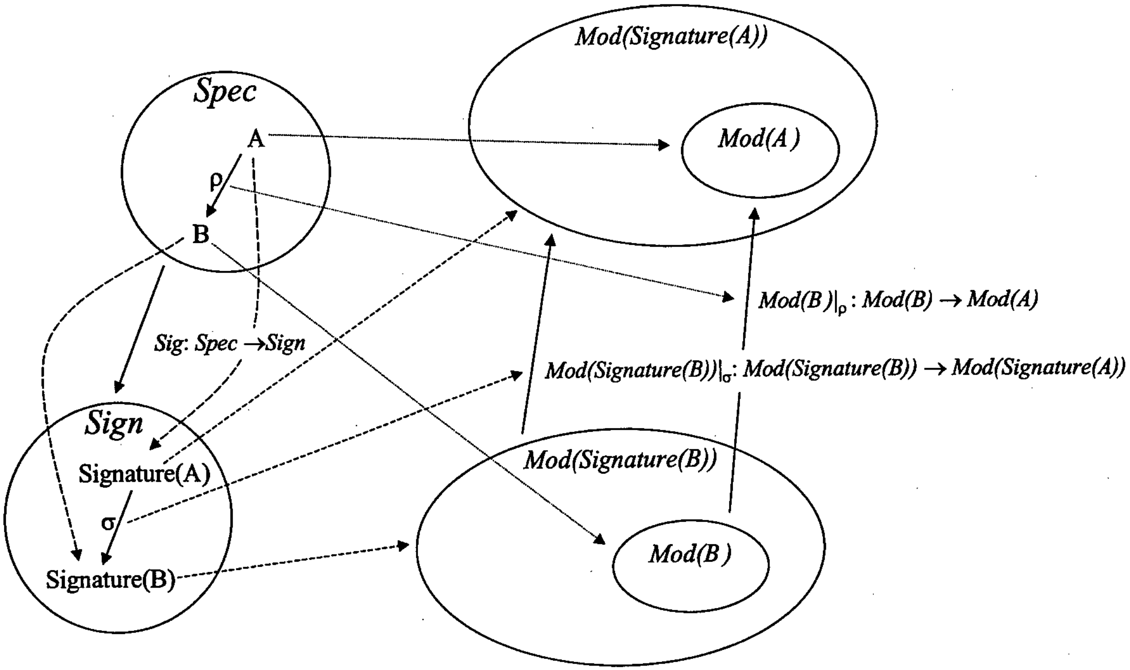


Figure B-8. Specification morphisms induce reduct functors

The existence of a specification model category reduct functor, ρ-reduct, associated with each specification morphism ρ:A→B, proves that all models of the codomain (target) of specification morphism ρ can be viewed as models of the domain (source) of ρ after removing the extra carrier sets and functions. Therefore, for a given specification morphism ρ:A→B, all specification models of B are also (after being ρ-reducted) specification models of A. It is this property that enables a specification morphism to be viewed as a design refinement since any implementation of the target specification is automatically an implementation of the source specification, but the reverse is not necessarily so.

Earlier in the appendix the claim was made that a design decision, expressed as additional properties in an algebraic specification, may reduce the number of possible implementations of a specification. Since the ρ-reduct of morphism ρ:A→B, is not necessarily surjective on the codomain restriction, i.e. the functor *Mod(B)*$|_\rho$*:Mod(B)→Mod(A)* is not epic, this means that there may be some *Mod(A)* models that are not covered by the models of *Mod(B)*. These models have effectively been eliminated by the design decisions that are implicit in the morphism from A to B.

Appendix C. Specification Development and Refinement

This appendix expands on the concepts introduced in Appendix B by describing how requirement specifications can be developed and refined into an implementation. Although the presentation and the proofs are original, the material has been published elsewhere.

A formal software development system that relies on algebraic specifications for requirements must have an approach for developing and refining large specifications. A specification can be developed by listing its sorts, operations and axioms, but it is difficult to specify large problems in such a monolithic fashion. A requirement specification can be refined (or at least its refinement indicated) by means of a specification morphism, but it is difficult to develop large, monolithic specification morphisms and target specifications for large requirement specifications.

The difficulty of developing and refining large specifications arises because one is trying to accomplish too much with each specification and each refinement. Breaking the problem into smaller pieces, developing specifications for them independently, and then combining the smaller specifications so that they specify the problem as a whole is one method of developing and refining large specifications. Large monolithic refinements can be dealt with by developing many smaller refinements for the smaller specifications (and doing less with each refinement) and then combining them in such a way that they refine the problem as a whole.

It is important that specification development methods be consistent with specification refinement methods. The way that specifications are extended and combined needs to be consistent and compatible with the way that refinements are extended and combined. For example, individual refinements of component specifications should aggregate to form a refinement of the aggregation of the component specifications.

This appendix is structured as follows. Section C.1 describes how an algebraic specification can be developed as a translation of an existing specification, or as an extension, conservative extension, or definitional extension of an existing specification. It also describes how models and model classes relate to and are related by these constructions. Section C.2 describes how large "structured" specifications can be built as an aggregate of other specifications using a colimit construction. Section C.3 describes the primary specification refinement mechanism, the interpretation, and how interpretations can be extended and composed. Section C.4 describes how refinements of component specifications can be combined to form a refinement of an aggregate specification.

C.1 Properties of Specification Morphisms

In this section we describe some of the additional restrictions that can be placed on specification morphisms and examine the effects they have on the models associated with the source and target specifications of the morphism.

C.1.1 Specification translation

The simplest manipulation of a specification is a signature translation. A signature translation enables one to change the sort and operation names. As long as the names remain distinct after translation, the class of models associated with a specification does not change.

Definition C.1. Specification B is a *translation* of specification A if there exists an epimorphism $\sigma{:}A \rightarrow B$ and *Axioms*(B) = $\{\boldsymbol{\sigma}(\alpha) \mid \alpha \in$ *Axioms*(A)$\}$, where $\boldsymbol{\sigma} = Sen_{\mathrm{Arrow}}(\sigma)$ is a function that translates A-sentences to B-sentences. The epimorphism is also called a translation. If the translation is an isomorphism we refer to it as a *conservative translation.*

The name translation is overloaded in that both the translation morphism and the codomain (target) of the translation morphism are called translations. A translation (morphism and target) can be indicated by the code in Figure C-1.

```
Spec List is
  sort List, E
  op Empty:          -> List
  op Append: E, List -> List
  op In:     E, List -> Boolean
  Constructors {Empty, Append} Construct List
  Forall s,s1,s2:List, e,e1,e2,e3,e4:E
  Ax Not(in(e,Empty))
  Ax in(e1,Append(e2,s)) ⇒ (e1=e2 ∨ in(e1,s))
  Ax (Insert(e1,s1) = Insert(e2,s2)) ⇔ (e1=e2 ∧ s1=s2)
  Ax (Insert(e1,Insert(e2,s)) = Insert(e3,Insert(e4,s)) ⇔ (e1=e3 ∧ e2=e4)
end-spec

Spec Queue is
  translate List by {List → Queue, Append → Enqueue}
```

Figure C-1. Specification translation

In Figure C-1, specification Queue is the translation of specification List by the signature morphism {List→Queue, Append→Enqueue}. (Specification Queue contains the sorts, operations and axioms of specification List translated so that the sort List is now sort Queue and the operation Append is now operation Enqueue.) There is an obvious isomorphism between specifications List and Queue, thus there is a conservative translation between the specifications. Conservative translations have no effect on the meaning of a specification in terms of its models; it is a simple renaming of the sorts and operations. Thus the class of models associated with the source specification of a conservative translation is identical to the class of models associated with the target specification.

A non-conservative translation (epimorphism only) could translate two distinct sorts or operations in the source specification to the same sort or operation in the target specification. A non-conservative translation can be viewed as a design decision that reduces the class of models by forcing some carrier sets or functions of all models to be identical where before they could be different from each other.

Proposition C.2. Translations and conservative translations are closed under composition. Therefore if f:A→B and g:B→C are (conservative) translation morphisms then their composition, $g \circ f$, is a (conservative) translation morphism.

$$\begin{tikzcd} A \arrow[r, "t"] \arrow[rr, dashed, bend right, "t"'] & B \arrow[r, "t"] & C \end{tikzcd} \qquad \begin{tikzcd} A \arrow[r, "ct"] \arrow[rr, dashed, bend right, "ct"'] & B \arrow[r, "ct"] & C \end{tikzcd}$$

Proof: Epimorphisms and isomorphisms are closed under composition in any category [AHS90]. □

C.1.2 Specification extension

An extension starts with an existing specification as a base and then adds to it additional sorts, operations and/or axioms. The same base specification could be extended in many different ways to form many different extended specifications. A specification extension is somewhat analogous to the #include macro in C, or like the "with" statement in Ada.

Definition C.3. Specification B is an *extension* of specification A iff there exists a monomorphism σ:A→B. The monomorphism is also referred to as an *extension*.

The usual method for constructing an extension involves literally extending a specification by adding more sorts, operations, and/or axioms to a specification to create a new specification, as is done in

Figure C-2. The semantics of the import statement are to import "textually" the sorts, operations and axioms of one specification into another. When extensions are created in this fashion the monomorphism is trivial to prove as each sort, op, and axiom is imported into the extended specification. Note, however, that the definition for an extension does not require it to be built this way, or to have the same sort and operation names or the same (translated) axioms as the specification being extended.

Example C.4. A specification extension (and an extension morphism) can be indicated by the code in Figure C-2. Note that the axiom "`Not(Insert(e,s) = Insert(e,Insert(e,s)))`" expresses a property that is not typically associated with sets, hence the name given to the specification. The text "`Import set`" means the sorts, operations and axioms of specification Set are textually imported into specification Set-No-Longer. By construction, an extension morphism exists between the Set and Set-No-Longer specifications, i.e. Set –e→ Set-No-Longer. The extension morphism is called an import morphism for obvious reasons, i.e. *import-morphism*:Set→ Set-No-Longer. The Set-No-Longer specification is referred to as an extension of the Set specification. Note also that the sorts Junk, More-Junk and the operation GIGO have no axioms describing their properties and are completely arbitrary extensions.

```
Spec Set-no-Longer is
  Import Set

  Sort Junk, More-Junk
  Op GIGO: Junk, Junk -> More-Junk

  Forall s:Set, e:E
  Ax Not(Insert(e,s) = Insert(e,Insert(e,s)))
end spec
```

Figure C-2. Extending a specification

Example C.5. A specification for a linear order can be extended to be either a discrete linear order or a dense linear order. The example in Figure C-3 comes from Chapter 4 of [Dim98]. Note that there is no morphism between the DiLO and DeLO specifications even though both specifications have extension (import) morphisms to them from specification LO.

If a morphism is viewed as a refinement mechanism, an extension can be viewed as a stricter form of refinement where all source operations and sorts are refined into separate target operations and refinements. An extension morphism preserves the structure of the source specification, in terms of the number of original sorts and operations, by not allowing them to be "collapsed" in the target specification. Thus all implementations of the target specification have separate and distinct carrier sets and functions for the sorts and operations that originate in the source specification.

```
spec LO            -- Linear Order
  sort E
  op 0: -> E
  op <: E, E -> Boolean

  Forall x,y,z:E
  ax Not(x < x)
  ax (x < y) ∨ (y < x) ∨ (x = y)
  ax (x < y) ∧ (y < z) ⇒ (x < z)
  ax Not(x < 0)
end-spec
```

```
spec Di-LO         -- Discrete Linear Order
  import LO
  Forall x:E
  ax ∃ y:E (∀ w:E ((x < y) ∧ Not((x < w) ∧ (w < y))))
end-spec
```

```
spec De-LO         -- Dense Linear Order
  import LO

  Forall x,y:E
  ax ∃ w:E (x < y) => (x < w) ∧ (w < y)
end-spec
```

Figure C-3. Linear-Order extended to Discrete/Dense-Linear-Order

C.1.3 Factoring a specification morphism

To be a specification morphism there needs to be a signature mapping from a source specification to a target specification and the (translated) source axioms must be theorems of the target. Now that there is a notion of the translation morphisms and the extension of morphisms, an arbitrary specification can be factored into component morphisms. A factored specification morphism separates the two concerns of "signature mapping" and "axioms as theorems" into two distinct simpler morphisms, a translation part and an extension part (that does not involve translation). The definition of a translated source specification and morphism factorization makes some later definitions and proofs easier.

Proposition C.6. Every specification morphism $\sigma{:}A\rightarrow B$ has associated with it a specification A^σ referred to as the *translated source specification* of σ, where σ is factored into a translation morphism $\sigma^T{:}A\rightarrow A^\sigma$

and a monomorphism *id-monic*:$A^\sigma \rightarrow B$ where *Sorts*(A^σ) $\subseteq$ *Sorts*(B), *Operations*(A^σ) $\subseteq$ *Operations*(B), and σ = *id-monic* $\circ$ σ^T, i.e. the following diagram commutes:

$$\begin{tikzcd}
A \arrow[r, "\sigma^T"] \arrow[dr, swap, "\sigma"] & A^\sigma \arrow[d, "id\text{-}monic"] \\
& B
\end{tikzcd}$$

Proof: Let σ:$A \rightarrow B$ be a specification morphism where $\sigma = <\sigma_S, \sigma_\Omega>$. Let the sorts, operations and axioms of A^σ be defined as follows:

Sorts(A^σ) $= \{\sigma_S(s) \mid s \in$ *Sorts*(A)$\}$,
Operations(A^σ) $= \{\sigma_\Omega(f) \mid f \in$ *Operations*(A)$\}$, and
Axioms(A^σ) $= \{Sen_{\mathrm{Arrow}}(\sigma)(\alpha) \mid \alpha \in$ *Axioms*(A)$\}$.

Let morphism $\sigma^T = <\sigma_S, \sigma_\Omega>$ be the same pair of functions in terms of their set mappings as the specification morphism σ except that the codomain object is A^σ instead of B. By construction, σ^T is a translation morphism, σ^T:$A \rightarrow A^\sigma$. Also by construction, *Operations*(A^σ) $\subseteq$ *Operations*(B), *Sorts*(A^σ) $\subseteq$ *Sorts*(B), therefore *id-monic*:$A^\sigma \rightarrow B$ is the identity mapping between sorts and operations (with B as the codomain object instead of A^σ). Thus, *id-monic* is a signature monomorphism. *Id-monic* is also a specification morphism as by $\vdash$ -translation *Axioms*(B) $\vdash_B \alpha$ for all $\alpha \in$ *Axioms*(A^σ) since that was the case for $\alpha \in$ Axioms(A). Finally, the diagram commutes as

$\sigma_S(s) = i_S(\sigma^T{}_S(s))$ and
$\sigma_\Omega(f) = i_\Omega(\sigma^T{}_\Omega(f))$
for all $s \in$ *Sorts*(A) and all $f \in$ *Operations*(A). □

Proposition C.7. An extension σ:$A \rightarrow B$ can be factored into a *conservative* translation morphism σ^T:$A \rightarrow A^\sigma$ and an extension *id-monic*: $A^\sigma \rightarrow B$ where A^σ is isomorphic to A (and *Sorts*(A^σ) $\subseteq$ *Sorts*(B) and *Operations*(A^σ) $\subseteq$ *Operations*(B)).

Proof: The translation σ^T is an epimorphism by construction. Since an extension is a monomorphism, the translation σ^T is also a monomorphism by construction. Therefore σ^T is an isomorphism, a conservative translation, and $A^\sigma \cong A$. □

Notation: For any extension σ:$A \rightarrow B$, the sorts of specification B that are in the translated source specification A^σ are referred to as the *original sorts*, and they *originate* in specification A. The sorts in

Sorts(B) – *Sorts*(A^σ) are referred to as the *extended sorts*, and they *originate* in specification B. The same nomenclature applies to *original* and *extended operations*. In Figure C-2 the extended sorts are Junk and More-Junk and the extended operation is GIGO.

There are several types of specification extensions that have special properties and special meaning when related to software requirement and design artifacts. Two of the most important are conservative extensions and definitional extensions.

C.1.4 Conservative extension

A conservative extension is an extension that adds to, but (in one sense) does not modify what already exists in the original specification. A conservative extension $\sigma:A\rightarrow B$ is a morphism where the underlying entailment relation over A-sentences is not altered in specification B. Thus the meaning of any sentence over the original signature of a specification is unchanged in terms of its truth or falsehood under entailment in any conservative extension of a specification.

Definition C.8. A morphism $\sigma:A\rightarrow B$ is a *conservative extension* iff σ is an extension morphism and *Axioms*(B) $\vdash_B \alpha$ iff *Axioms*(A^σ) $\vdash_A \alpha$ for all $\alpha \in A^\sigma$-sentences where $A^\sigma$is the translated source spec of σ.

For any specification morphism, σ, we have proven that all models of the target specification can be reduced by the σ-reduct functor to become models of the source specification. However, if σ is also a conservative extension, the converse is also true. If there exist models of the target specification at all, then any model of the source specification can be extended (possibly in many ways) to become a model of the target specification.

Example C.9. A specification for a discrete linear order can be extended in a conservative fashion so that it has a non-decreasing function, "up". (Function "up" may increase the value given it or return the same value.) Any model of specification Di-LO can be extended with a function "up" that is a model of specification Di-LO-Up. This example comes from Chapter 4 of [Dim98].

```
spec Di-LO-Up
  import Di-LO

  op up: E -> E

  forall x,y:E
  ax (x < y) => x < up(y)
end-spec
```

One way to ensure that an extension is conservative is to extend a specification with sorts, operations and axioms that make no reference to the original sorts and operations. This is an unreasonable restriction as in many cases one extends a specification by building upon the existing sorts and operations of the original specification as was done in Example C.10. Unfortunately there is no simple syntactic guarantee that a new axiom that references the original sorts and operations does not alter the meaning of the original sorts and operations. One can, however, use the following syntax to indicate that one intends to extend a specification conservatively:

```
Spec Set-of-value-pairs is
Protect Set

Sort value
Op Left:  E → value
Op Right: E → value
end-spec
```

In the example above, the Set specification has been conservatively extended (as indicated by the text "`Protect Set`") with operations that return a value given an element of sort E. The new sort and operations are not defined with axioms and they do not affect the entailment relation over the sentences of the original Set specification. Thus, there is a conservative extension morphism between the specifications, i.e. Set $\xrightarrow{c}$ Set-of-value-pairs.

Conservative extensions can be related to programming in a programming language. When programming with a language such as Ada or FORTRAN, the built in data types and subprograms can be augmented with other data types and subprograms. One can combine two or more collections of unrelated data types and subprograms together. One can also create structured data types or subtypes from existing data types and combine subprograms with control structures to create more powerful subprograms. Conservative extensions and programming extensions differ in that with (normal) programming extensions a programmer does not have to worry that the use of a data type or subprogram will indelibly change its meaning. It simply isn't possible to do this when *using* a programming language construct. One can, using syntactically different methods, modify or overwrite existing programming language concepts, such as redefining a method in object-oriented programming, but in most programming languages it is simple to syntactically differentiate when one is using a type or subprogram or redefining (modifying or overwriting) it and programmers do not confuse the two.

For extensions of specifications, however, this stability of meaning is not given. The axioms of a specification logically relate the operations referenced within it and do not differentiate which operations should be fixed in meaning and which ones are being defined. Unlike a programmer, a specifier needs to prove that the extensions that are intended to be conservative do not change the underlying meaning of any of the existing sorts and operations. For specifications there is no simple syntactic check or guarantee that extensions preserve meaning as there is with programming languages.

Example C.10. A specification for a discrete linear order can be extended in a seemingly conservative fashion so that it has a maximum value function.

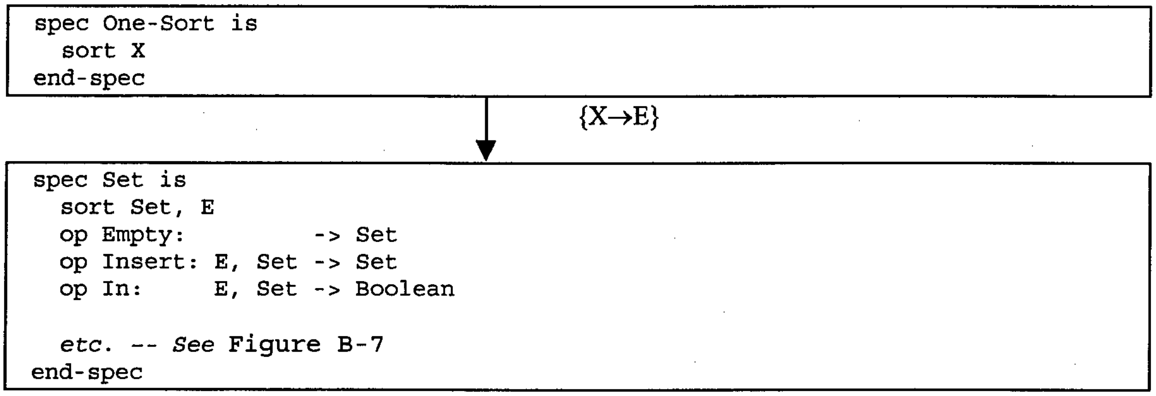
```
Spec LO-Max
Protect LO

  Op Max: -> E

  forall x:E
  ax x < Max or X = Max
end-spec
```

Despite the stated intentions of the syntax, "`Protect LO`", the extension is not conservative. The extended axiom has a "side effect" in that it implies that there is an upper bound in the discrete linear order. This means an axiom describing the property of having an upper bound,

"∀ x:E, ∃ y:E | x < y ^ x = y",

is a theorem of specification LO-Max but it is not a theorem of specification LO. Since that property can be expressed entirely in terms of specification LO's signature, the extension is not a conservative extension.

Conservative extensions are important in a software-engineering environment in that they represent an "embedding" of all the models of one specification in the models of another specification. All of the models of the specification being extended have one or more "model extensions" that are models of the extended specification. For a conservative extension *c*:A→B, all (abstract) implementations of specification B contain an "embedded" model of specification A and unlike with a normal extension, all models of specification A are represented in such a manner (the reduct is surjective). An arbitrary (non-conservative) morphism can be viewed as a refinement mechanism where some models have been eliminated in the models associated with the target specification (actually eliminated in the morphism reduct of the models of the target specification). A conservative extension morphism can be viewed as an "embedding" mechanism as every model can be extended with additional carrier sets and operations (sometimes in many different ways) so that it is a model of the target specification.



```
spec One-Sort is
  sort X
end-spec
```

```
spec Set is
  sort Set, E
  op Empty:          -> Set
  op Insert: E, Set -> Set
  op In:     E, Set -> Boolean

  etc. -- See Figure B-7
end-spec
```

Figure C-4. Conservative extension (parameterization)

Example C.11. The morphism One-Sort $-c\rightarrow$ Set, depicted in Figure C-4, is also an example of a conservative extension. Any model of specification One-Sort can be extended with set-like operations. (Any carrier set associated with sort X in specification One-Sort can be extended with a carrier set for the sort Set where the functions Empty, Insert, and In operate as one would expect according to the axioms of specification Set.) This corresponds to the intuitive notion that given any collection of elements, one can form a set of some subset of those elements.

A conservative extension morphism is important to the concept of parameterization as Example C.11 demonstrates. Current parameterization research is presented in Section 2.3 and Appendix E. This dissertations contributions to parameterization is the focus of Section 3.5, Section 4.2 and Chapter 5, where conservative extensions play an important part.

C.1.5 Definitional extension

A definitional extension is a conservative extension where the added sorts and operations are fully characterized over the original sorts and operations. A definitional extension cannot merely extend a specification with unrelated sorts and operations as is possible with a conservative extension. The new sorts and operations in a definitional extension must be defined and fully characterized (directly or indirectly) over the sorts and operations of the specification being extended.

Definition C.12. A morphism σ:A→B is a *definitional extension* (d-morphism) iff it is a conservative extension morphism and all extended sorts are constructed sorts and all extended operations are defined to be total and functional.

If σ:A→B is a definitional extension then models of specification A can be uniquely extended so that they are models of specification B. Because of the additional constraints put on by definitional extensions the extension that takes a particular model of specification A to a model of specification B is unique [BGG94].

Definition C.13. *Total and functional operations* are defined as follows:

if A is a specification and *f* is an operation with rank $s_1, s_2, \ldots s_n \rightarrow s$ then *f* is total and functional iff the following axioms are theorems of specification A:

$$\forall\ \alpha_1{:}s_1,\ \alpha_2{:}s_2,\ \ldots,\ \alpha_n{:}s_n,\ \exists\ y{:}s \mid f(\alpha_1, \alpha_2, \ldots, \alpha_n) = y$$
$$\forall\ \alpha_1{:}s_1,\ \alpha_2{:}s_2,\ \ldots,\ \alpha_n{:}s_n,\ y_1{,}y_2{:}s,\ \ (f(\alpha_1, \alpha_2, \ldots, \alpha_n) = y_1 \ \wedge\ f(\alpha_1, \alpha_2, \ldots, \alpha_n) = y_2) \Rightarrow (y_1 = y_2)$$

For models of specification A, the function associated with a total and functional operation *f* is guaranteed to return a unique output element of the output-sort carrier-set given input elements of the appropriate carrier-sets. Once the carrier-sets are fixed there can only be one function (in the sense of an abstract input/output relation) associated with the operation *f* as that function is defined to be unique by the axioms of specification A.

In a definitional extension, the axioms used to fully indicate the total and functional nature of an extended operation must, by definition, be conservative. Since the axiom's sole purpose is to fully define an operation, and they do not affect the meaning of any other sort or operation, they can be linked to that operation using the syntax indicated in Figure C-5, which is slight variation of the Specware language [SLM98]:

The text "`define f by`" in Figure C-5 can be viewed as a syntactic sugar replacement for additional axioms concerning operation *f* that confirm that operation *f* is functional and total. For example, if the rank of *f* is *f*: X→Y then the following axioms insure that *f* is functional and total:

$$\forall\ x{:}X,\ \forall\ y1{,}y2{:}Y, f(x) = y1 \wedge f(x) = y2 \Rightarrow y1 = y2 \qquad\qquad \forall\ x{:}X,\ \exists\ y{:}Y \mid f(x) = y$$

Axioms such as these need to be theorems of the specification for each defined operation in order for the operation to be proven both functional and total.

```
Spec Augmented-List is
  Protect List

  Op No-Dupes: List → Boolean
  Op Add-if-new: E, List → List
  Op Count: E, List → Nat
  Op Perm: List, List → Boolean

Forall e,e1,e2:E, l, l1,l2:List

  define No-Dupes by
    Ax No-Dupes(Empty)
    Ax No-Dupes(Insert(e,l)) = (No-Dupes(l) ∧ not(In(e,l)))

  define Add-if-new by
    Ax Add-if-new(e, Empty) = Append(e, Empty)
    Ax Add-if-new(e, l) = if in(e, l)then l
                                   else Append(e, l)

  define Count by
    Ax Count(e, Empty) = 0
    Ax Count(e1,Append(e2, l)) = if e1=e2 then 1 + Count(e1,l)
                                          else Count(e1,l)

  define Perm by
    Ax Perm(l1,l2) = (count(e,l1) = count(e,l2))

end-spec
```

Figure C-5. Total and functional definitions

For specification Augmented-List, the operations No-dupes, Add-if-new, and Count are defined over the original operations, the axioms in the definition clause are conservative (they do not change the entailment relation over the original signature), and the axioms insure that these operations are functional and total. Note that the operation Perm is defined using the newly defined Count operation and therefore it is indirectly defined over the original specification. Because all extended operations in Figure C-5 are defined to be functional and total, Augmented-List is a definitional extension of List, i.e. List$-_{d}\rightarrow$Augmented-List.

All extended sorts that are definitional extensions must be constructed from the existing sorts. Examples of this are subsorts (s | *f*), quotient sorts (s / *f*), product sorts(s_1, s_2, …), and coproduct sorts ($s_1 + s_2 + \ldots$), [SLM98].

Definition C.14. Given a set, *Sorts*(A), of existing sorts, a collection of *Constructed sorts*, CS, is defined inductively as follows:

- All original sorts are constructed sorts. s ∈ *Sorts*(A) implies that s ∈ CS.

- All subsorts are constructed sorts. $s \in CS$ implies that $s|f \in CS$ where f:s→boolean and $\alpha \in s|f$ iff $\alpha \in s \wedge f(\alpha)$. All subsorts have an implicit relaxation function that takes elements of $s|f$ to s, i.e. (relax f): $s|f \rightarrow s$.
- All quotient sorts are constructed sorts. $s \in CS$ implies that $s/f \in CS$ where f:s,s→boolean is an equivalence relation (reflexive, transitive, and symmetric) and $\alpha \in s/f$ iff α is an equivalence class induced on s by f. All quotient sorts have an implicit quotient function that takes elements of s to their quotient (equivalence class) in s/f, i.e. (quotient f): $s \rightarrow s/f$.
- All product sorts are constructed sorts. $s_1, s_2, \ldots s_n \in CS$ implies that $(s_1, s_2, \ldots s_n) \in CS$ where $(s_1, s_2, \ldots s_n)$ is the cross product of sort values. All product sorts have a family of implicit projection functions that extract a sort value from a tuple, i.e. (project i): $s_1, s_2, \ldots s_n \rightarrow s_i$.
- All coproduct sorts are constructed sorts. $s_1, s_2, \ldots s_n \in CS$ implies that $s_1 + s_2 + \ldots + s_n \in CS$ where + is analogous to the disjoint union of sort values. All coproduct sorts have a family of implicit embedding functions that embed a sort value into a coproduct, i.e. (embed i): $s_i \rightarrow s_1 + s_2 + \ldots + s_n$.

For models of specification A, the carrier set associated with a constructed sort s is defined based on the carrier sets associated with the sorts from which it was constructed. Once the carrier-sets of the defining sorts are fixed (and the defining functions are fully defined) there can only be one carrier set associated with the constructed sort s.

Constructed sorts are related to programming language types in the following way. Subsorts are similar to subtypes, product sorts are similar to record types and coproduct sorts are similar to polymorphic types or variant records. Few programming languages have a construct similar to quotient sorts; however, they can be simulated by using an equivalence relation over a given type in place of using the "=" (strict equality) operation. One can add axioms to a specification to indicate that the relationships defined above hold between the original sorts and the constructed sorts, or one can use the sort axioms in Figure C-6 as the semantic equivalence of those axioms.

The text "`sort-axiom S =`" in Figure C-6 can be viewed as a syntactic sugar replacement for additional axioms that constrain the sort being defined by relating it to an existing sort and some fully defined function(s). The sort-axiom also serves as a placeholder for the definition of the polymorphic access operations (such as relax, quotient, project, embed described in Definition C.14). In the example above, the extended sorts, List-2, SubsortList, QuotientList, ProductList, and CoproductList are all constructed sorts. Note that ProductList is constructed using other constructed sorts. Because the extended sorts are all fully defined there is a definitional extension morphism from specification Augmented-List to specification Augmented-List-2.

```
 Spec Augmented-List-2 is
   Protect Augmented-List

   Sort List-2, SubsortList, QuotientList, ProductList, CoproductList
   Sort-axiom List-2        = List
   Sort-axiom SubsortList   = List|No-Dupes
   Sort-axiom QuotientList  = List/Perm
   Sort-Axiom ProductList   = List, QuotientList, SubsortList
   Sort-Axiom CoproductList = E + List
end-spec
```

Figure C-6. Constructed sorts

The concept of the extension being built using the original specification carries over to the models of the specifications as well. For a definitional extension σ:A→B the σ-reduct functor from Mod(B) to Mod(A) is epic and monic and has an inverse functor that is the unique extension taking models of Mod(A) to models of Mod(B) [BGG94]. Each carrier set of an extended sort of a model in Mod(B) is related to the carrier sets in some model in Mod(A) in some unique way, i.e. they are related by being the same carrier set, a particular subset, quotient set, product of sets, or coproduct of sets. Every function associated with an extended operation of a model in Mod(B) has a fully defined meaning by Definition C.13; thus for any given Mod(A) model there is only one function that fits the defined meaning that is in the associated model in Mod(B).

Definitional extensions are important in a software-engineering environment in that they represent constructing a "larger" model (one with more carrier sets and functions) from an existing model in a unique way. For a definitional extension *d*:A→B, every (abstract) implementation of specification A has a unique extension that makes it an (abstract) implementation of specification B. This is unlike a conservative extension where there may be many such extensions. Since every model of specification A has a single extension that makes it a model of specification B, we can view a definitional extension as an (abstract) implementation constructor. Given an (abstract) implementation of specification A, we can construct an abstract implementation of specification B in a single way. (There may be more then one way to code the implementation but their abstract functionality is identical.) Thus while we view a conservative extension as an implementation "embedding" mechanism, a definitional extension is an implementation "construction" mechanism. Section C.3, which covers specification refinement, has several examples of definitional extensions.

C.1.6 Composition of extension morphisms

An extension ensures that the target specification does not collapse any of the distinct sorts and operations of the source specification. A conservative extension represents an embedding or composition of independent models. A definitional extension represents a construction of implementations. We would expect such properties to compose.

Proposition C.15. Extensions, conservative extensions and definitional extensions are closed under composition, i.e. if f:A→B and g:B→C are both extensions, conservative extensions or definitional extensions then their composition, $g \circ f$, is an extension, conservative extension or definitional extension respectively.

$$\begin{tikzcd} A \arrow[r, "e"] \arrow[rr, dashed, bend right, "e"'] & B \arrow[r, "e"] & C \end{tikzcd} \qquad \begin{tikzcd} A \arrow[r, "c"] \arrow[rr, dashed, bend right, "c"'] & B \arrow[r, "c"] & C \end{tikzcd} \qquad \begin{tikzcd} A \arrow[r, "d"] \arrow[rr, dashed, bend right, "d"'] & B \arrow[r, "d"] & C \end{tikzcd}$$

Proof:

<u>Extensions</u>: Given extension morphisms f:A→B and g:B→C, both f and g are monomorphisms, therefore their composition, $g \circ f$, is a monomorphism (and hence an extension).

<u>Conservative Extensions</u>: Given conservative extension morphisms f:A→B and g:B→C, their composition, $g \circ f$, is an extension morphism by the proof above. By f, *Axioms*(B) $\vdash_B \alpha$ for each $\alpha \in A^f$-sentences iff *Axioms*(A^f) $\vdash_A \alpha$ where A^f is the translated source specification of f. By g, *Axioms*(C) $\vdash_C \beta$ for each $\beta \in B^g$-sentences iff *Axioms*(B^g) $\vdash_B \beta$ where B^g is the translated source specification of g. Thus by the transitivity and $\vdash$-translation properties of entailment for $g \circ f$, *Axioms*(C) $\vdash_C \alpha$ for each $\alpha \in A^{g \circ f}$-sentences iff *Axioms*($A^{g \circ f}$) $\vdash_A \alpha$ where $A^{g \circ f}$ is the translated source specification of $g \circ f$.

<u>Definitional Extensions</u>: Given definitional extension morphisms f:A→B and g:B→C, their composition, $g \circ f$, is a conservative extension morphism by the proof above. By f, the extended sorts in specification B are constructed sorts. By g, the extended sorts in specification C are constructed sorts. Therefore all sorts in specification C that did not originate in specification A are constructed sorts. Similar logic holds for the

extended operations in specifications B and C. Since $g \circ f$ is a conservative extension where all extended sorts in the codomain are constructed sorts and all extended operations in the codomain are total and functional, $g \circ f$ is a definitional extension. □

Example C.16. Using the syntax for sort-axioms and the define-by statement described above for definitional extensions it is easy to see how d-morphisms compose from a syntactic perspective. The definitional extensions List→Augmented-List and Augmented-List→Augmented-List-2 can be composed such that there is a definitional extension List→Augmented-List-2. The composite morphism (and specification) can be represented by syntactically composing the extensions.

C.2 Specification Aggregation

The previous section described how a larger specification can be constructed by adding additional sorts, operations and axioms to an existing specification. In this section an aggregation mechanism is described that enables existing specifications to be combined to create a larger structured specification. The specification is structured in that the specifications that serve as components of the larger specification can be identified as such and the aggregation mechanism documents how these component specifications are combined to form the larger specification and are related to the larger specification.

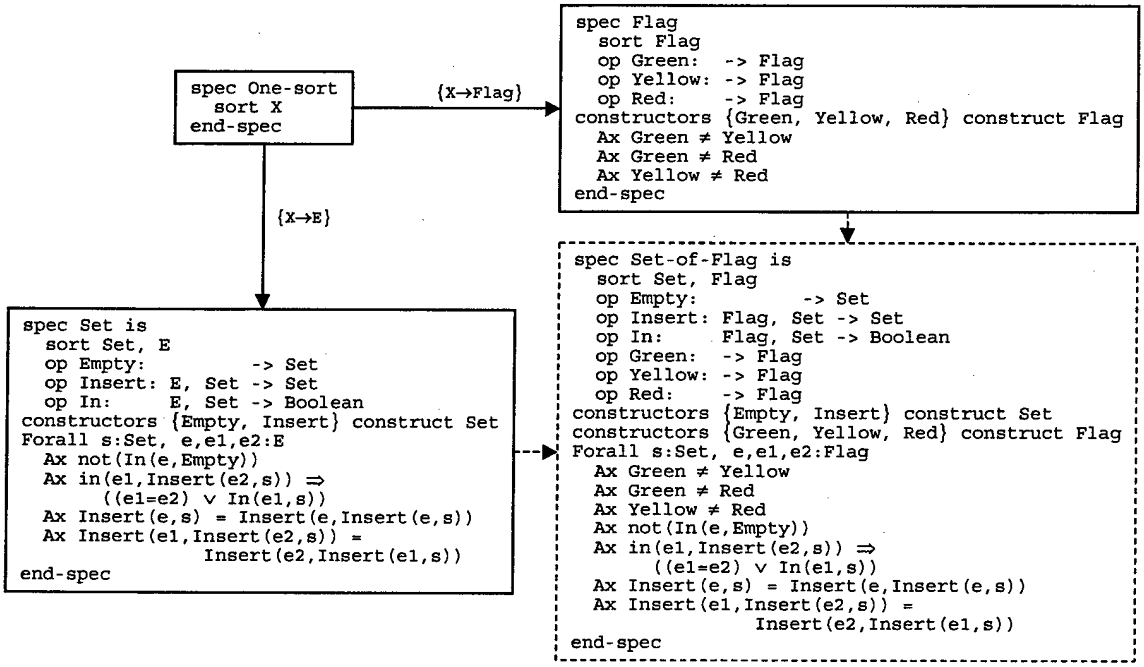
Aggregation enables many smaller specifications to be written and then combined to specify larger problems. When creating an aggregate, the aggregate object should be minimal in that it should only include those properties (requirements) that are present in the component specifications. The process for creating an aggregate should not place any additional constraints on the problem being specified beyond those that can be derived from the components of the aggregate.

C.2.1 Pushout

In the category *Spec*, a pushout, and the more general colimit, are category constructions that define a minimal aggregate object from a collection of related objects. The minimal aggregate specification represents the minimal requirement specification. Any other aggregate specification is guaranteed to have more structure or properties (and hence more requirements) than can be inferred from the component specifications.

A pushout in the category *Spec* has the same structure as depicted in Figure A-8. However, the objects are specifications and the arrows are specification morphisms. Specification A represents the signature and properties over the signature that must be present (after translation and by entailment) in both specifications B1 and B2 before they can be merged.

Example C.17. The Set specification can be combined with a specification for an enumerated type, Flag, to form the specification Set-of-Flag. Figure C-7 provides a graphical depiction of the commuting pushout square. Note that the purpose of the One-Sort specification is to indicate how the other specifications are to be related.



```
spec One-sort
  sort X
end-spec
```

```
spec Flag
  sort Flag
  op Green:  -> Flag
  op Yellow: -> Flag
  op Red:    -> Flag
constructors {Green, Yellow, Red} construct Flag
  Ax Green ≠ Yellow
  Ax Green ≠ Red
  Ax Yellow ≠ Red
end-spec
```

```
spec Set is
  sort Set, E
  op Empty:          -> Set
  op Insert: E, Set -> Set
  op In:     E, Set -> Boolean
constructors {Empty, Insert} construct Set
Forall s:Set, e,e1,e2:E
  Ax not(In(e,Empty))
  Ax in(e1,Insert(e2,s)) ⇒
       ((e1=e2) v In(e1,s))
  Ax Insert(e,s) = Insert(e,Insert(e,s))
  Ax Insert(e1,Insert(e2,s)) =
                 Insert(e2,Insert(e1,s))
end-spec
```

```
spec Set-of-Flag is
  sort Set, Flag
  op Empty:            -> Set
  op Insert: Flag, Set -> Set
  op In:     Flag, Set -> Boolean
  op Green:  -> Flag
  op Yellow: -> Flag
  op Red:    -> Flag
constructors {Empty, Insert} construct Set
constructors {Green, Yellow, Red} construct Flag
Forall s:Set, e,e1,e2:Flag
  Ax Green ≠ Yellow
  Ax Green ≠ Red
  Ax Yellow ≠ Red
  Ax not(In(e,Empty))
  Ax in(e1,Insert(e2,s)) ⇒
       ((e1=e2) v In(e1,s))
  Ax Insert(e,s) = Insert(e,Insert(e,s))
  Ax Insert(e1,Insert(e2,s)) =
                 Insert(e2,Insert(e1,s))
end-spec
```

Figure C-7. Simple pushout example: Set-of-Flag

Proposition C.18. Given specification monomorphisms f: A→B1 and g : A→B2 where $f = <f_S, f_\Omega>$ and $g_S = <g_S, g_\Omega>$, and assuming that the sets of sort and operation names of the three specifications are disjoint, the pushout specification C can be constructed in the following manner:

Let specification Σ1 be a conservative translation of specification B1 where the names of the sorts and operations of B1 have been translated to the names of the specification A sorts and operations as indicated by the morphism f:A→B1, let the B1 axioms be translated in accordance with the new sort and operation names

i.e. $\Sigma 1 = <(\textit{Sorts}(\text{B1}) - \{f_{S(\alpha)} \mid \alpha \in \textit{Sorts}(\text{A})\}) \cup \textit{Sorts}(\text{A}),$
$(\textit{Operations}(\text{B1}) - \{f_{\Omega(\alpha)} \mid \alpha \in \textit{Operations}(\text{A})\}) \cup \textit{Operations}(\text{A}),$
{B1 axioms translated from B1-sentences to sentences over the sort and operation names above}>

Let specification $\Sigma 2$ be similarly derived from specification B2,

i.e. $\Sigma 2 = <(\textit{Sorts}(\text{B2}) - \{g_{S(\alpha)} \mid \alpha \in \textit{Sorts}(\text{A})\}) \cup \textit{Sorts}(\text{A}),$
$(\textit{Operations}(\text{B2}) - \{g_{\Omega(\alpha)} \mid \alpha \in \textit{Operations}(\text{A})\}) \cup \textit{Operations}(\text{A}),$
{B2 axioms translated from B2-sentences to sentences over the sort and operation names above}>

Let the sorts, operations and axioms of the pushout specification C be defined as follows:

$$\begin{aligned} \textit{Sorts}(\text{C}) &= \textit{Sorts}(\Sigma 1) \cup \textit{Sorts}(\Sigma 2) \\ \textit{Operations}(\text{C}) &= \textit{Operations}(\Sigma 1) \cup \textit{Operations}(\Sigma 2) \\ \textit{Axioms}(\text{C}) &= \textit{Axioms}(\Sigma 1) \cup \textit{Axioms}(\Sigma 2) \end{aligned}$$

and the morphisms from B1 and B2 to C be the obvious ones based on the relationships between the Bx, Σx, and C specifications, i.e. the compositions of B1→$\Sigma 1$→C and B2→$\Sigma 2$→C.

Proof: In order to prove the above construction is valid we must prove that it results in a commuting square where the constructed pushout specification has a unique arrow to the terminal specification of any other commuting square.

Existence of pushout object: By this method of construction, the B1 and B2 specifications are translated to $\Sigma 1$ and $\Sigma 2$ based on the unifying sort and operation names in specification A and the morphisms A→B1 and A→B2. Essentially our construction has created the conservative translation morphisms B1→$\Sigma 1$ and B2→$\Sigma 2$ where $\Sigma 1$ and $\Sigma 2$ share the signature of specification A. The pushout specification C is merely a union of the sets of sorts, operations, and axioms of $\Sigma 1$ and $\Sigma 2$. The monomorphisms $\Sigma 1$→C and $\Sigma 2$→C are the identity mappings of the sort and operation names where the axioms are trivially proven to be theorems of specification C. The axioms of specification A do not need to be included in C as they are theorems (after translation) of specifications B1 and B2 and hence are theorems of specification C. The composite morphisms A→B1→C and A→B2→C are equivalent by construction as the sorts and operations of specification A are mapped (via $\Sigma 1$ for B1 and via $\Sigma 2$ for B2) to the same sorts and operations of specification C, therefore the pushout square is a commutative square.

Existence of Arrow: Assume that there exist other C-arrows h:B2→D and j:B1→D such that that $j \circ f = h \circ g$. Let $\sigma 1$:$\Sigma 1$→B1 and $\sigma 2$:$\Sigma 2$→B2 be the conservative translations where $\sigma 1 = <\sigma 1_S, \sigma 1_\Omega>$ and $\sigma 2 = <\sigma 2_S, \sigma 2_\Omega>$. The signature morphism from k:*Signature*(C)→*Signature*(D) is defined as follows: Let k

be a signature morphism, $k = \langle k_S, k_\Omega \rangle$ where $k_S = \{ k_s \mid k_s = h_s \circ g_s$ (or equivalently $j_s \circ f_s$) if s ∈ Sorts(A) otherwise, $k_s = h_s \circ \sigma2_S$ if s ∈ Sorts(Σ2) OR $k_s = j_s \circ \sigma1_S$ if s ∈ Sorts(Σ1)$\}$ and where k_Ω is similarly defined over the operations in specifications A, Σ1, and Σ2 and the operation mapping functions from those specifications to specification D. By this construction we know there exists a signature morphism from C to D. The signature morphism Signature(C)→Signature(D) is a specification morphism as it is a given that the translated axioms in *Axioms*(B1) and *Axioms*(B2) are theorems of specification D and therefore the axioms in *Axioms*(C) can be similarly translated and proved as theorems.

$$\begin{tikzcd}
 & B1 \arrow[ddd, dashed, "g'"'] \arrow[dr, dotted] \arrow[ddddl, bend right, "j"] & & A \arrow[ll, "f"'] \arrow[ddd, "g"] \\
 & & \Sigma1 \arrow[ul, dashdotted, "\sigma1"'] \arrow[ddl, dotted] & \\
 & & \Sigma2 \arrow[dl, dotted] \arrow[dr, dashdotted, "\sigma2"] & \\
 & C \arrow[dl, dotted, "k"'] & & B2 \arrow[ll, dashed, "f'"] \arrow[ul, dotted] \arrow[dlll, bend left, "h"] \\
D & & &
\end{tikzcd}$$

Figure C-8. Diagram supporting the existence of arrow C→D

Arrow Completes a Commuting Diagram: The sorts and operations in specification C can be split into three groups, those originating from specification A, those originating in specification B1 (and not A), and those originating in specification B2 (and not A). In each case, the mapping of these three groups of sorts and operations of specification C to specification D is defined to be the same as the mapping of these sorts and operations from their non-C origin. Thus by construction, the arrow completes a commuting diagram.

Uniqueness of Arrow: Let *m* be any other specification morphism, *m*:C→D for which Figure A-8c forms a commuting diagram. Assume that morphism *m* is a different arrow then *k*, i.e. $m_S \neq k_S$ or $m_\Omega \neq k_\Omega$. Since the arguments for sorts and operations are identical, lets work with sorts only. If $m_S \neq k_S$ then there exists an s ∈ Sorts(C) such that $m_S(s) \neq k_S(s)$. However, based on our assumption for *m* we know that the overall diagram commutes for both *m* and *k*, i.e. the arrows A→C, B1→C, and B2→C when composed with arrows *k*:C→D and *m*:C→D form equivalent arrow pairs. For the composed arrows to be equivalent and there exist a sort s ∈ Sorts(C) such that $m_S(s) \neq k_S(s)$ then s must be a sort in C that is not mapped to by any of the morphisms A→C, B1→C, and B2→C. However, by construction we know that this is not the case, as all sorts in specification C are mapped to by at least one sort in specification A, B1, or B2. Therefore by construction we know there does not exist an s $\in$ *Sorts*(C) such that $m_S(s) \neq k_S(s)$. By similar reasoning we know that there does not exist an $f \in$ *Operations*(C) such that $m_\Omega(f) \neq k_\Omega(f)$. By this contradiction, we know that our original assumption is wrong and $m = k$. □

Alternative pushout construction methods use the B1 (or B2) sort and operation names as the unifying names. Unfortunately, all of these construction methods rely on the sort and operation names of the three specifications being disjoint and that the specification morphisms be monomorphisms.

The construction method can be extended to work with name spaces that are not disjoint by first translating the specifications so the sort and operation names are disjoint. Disjoint names can be arbitrarily chosen, or they can be constructed by appending the specification name (or some unique identifier) to all the sort and operation names of each specification, i.e. using the sort or operation "specification of origin" to distinguish it from other sorts and operations.

To extend the pushout construction method of Proposition C.18 to work with non-monomorphisms one must be able to indicate when the original specification morphisms (f and g) have merged two or more sorts (or operations). In the original construction the specification B1 (and B2) sort and operation names were translated to their specification A names. However, with non-monomorphisms there are two (or more) possible B1 (B2) names to choose from. An equivalence relation is used where the sorts (or operations) that belong to the same equivalence group are those that have been identified by the signature mapping as being "merged". Thus if sorts s_1 and s_2 of specification A are mapped to the same sort, β, in specification B1 then sorts s_1 and s_2 form an equivalence group sort in specification $\Sigma 1$ and the B1 sort β is translated to that equivalence group name. Note that specification $\Sigma 1$ will still be a conservative translation of specification B1 and hence an isomorphism. The construction for the sorts and operations of the pushout specification C is now a merging of the equivalence relations instead of a simple union of sets. If sorts s_1 and s_2 form an equivalence group sort in specification $\Sigma 1$ and sorts s_2 and s_3 form an equivalence group sort in specification $\Sigma 2$ then s_1, s_2, and s_3 form a single equivalence group sort in the pushout specification C. The axioms of specifications $\Sigma 1$ and $\Sigma 2$ (B1 and B2) are then translated using the "names" of the new equivalence group sorts and equivalence group operations.

The proof that the extended constructions above, which remove the disjoint name and monomorphism restrictions, will still result in a pushout specification is left as an exercise for the reader.

Note that the pushout specification constructed by the method defined in Proposition C.18 (and its extensions) is not unique. There are many other suitable specifications that are isomorphic to the one constructed by the method above. Any of these isomorphic specifications could serve as the pushout specification. The names given to the pushout specification's sorts and axioms are completely arbitrary (as long as they are distinct and do not cause further "merging"). Also, any set of axioms that has the same (isomorphic) theory in its closure can be used. In fact any specification that is isomorphic to the one constructed could be used as the pushout specification. For example, the (translated) axioms of specification A could be added to the pushout specification C as they are guaranteed to be theorems of the specification. Any other set of axioms that generate the same theory as the axioms we constructed for specification C could also serve as axioms for the pushout specification.

C.2.2 Pushouts in *Spec* preserve morphism properties

In Definition A.15 a pushout was defined in terms of arbitrary specification morphisms. What if the specification morphisms were one of the extension morphisms? How would that effect the constructed morphisms?

Proposition C.19. The pushout of a (conservative, definitional) extension morphism along a specification morphism results in a (conservative, definitional) extension morphism.

$$\begin{tikzcd}
A \arrow[r, "e"] \arrow[d] & B1 \arrow[d, dashed] & & A \arrow[r, "c"] \arrow[d] & B1 \arrow[d, dashed] & & A \arrow[r, "d"] \arrow[d] & B1 \arrow[d, dashed] \\
B2 \arrow[r, dashed, "e"] & C & & B2 \arrow[r, dashed, "c"] & C & & B2 \arrow[r, dashed, "d"] & C
\end{tikzcd}$$

Proof: The construction method from Proposition C.18 (and its extensions) are used in this proof. As any other pushout specifications and pushout morphisms will be isomorphic to the ones constructs and properties like the extensions are preserved under isomorphic changes it is safe to reason from the particular to the general using our proposed pushout construction method.

Extensions: For purposes of contradiction, given an extension morphism *f*:A→B1 and morphism *g*:A→B2, assume that the pushout of *f* along *g*, *f* ':B2→D, is not a monomorphism (extension). If *f*', where

$f' = \langle f'_S, f'_\Omega \rangle$ is not a monomorphism then there must exist s_1, $s_2 \in$ *Sorts*(B2) such that $s_1 \neq s_2$ and $f'_S(s_1) = f'_S(s_2)$ or there must exist $f_1, f_2 \in$ *Operations*(B2) such that $f_1 \neq f_2$ and $f'_\Omega(f_1) = f'_\Omega(f_2)$. Since the arguments are the same we will work with the sorts s_1 and s_2. It is given that f is an extension (monomorphism). By the construction method (and its extensions) the sorts in the conservative translation $\Sigma 1$, which is based on f, will have no equivalence group sorts. Also, by the construction method, if s_1, $s_2 \in$ *Sorts(*B2*)* where $s_1 \neq s_2$, then s_1 and s_2 remain different sorts in the conservative translation to specification $\Sigma 2$. This means that when the (translated) sorts s_1 and s_2 are mapped to the sorts in specification C they will remain separate sorts as the $\Sigma 1$ specification contains no equivalence group sorts that would cause the $\Sigma 2$ (B2) sorts to be merged. By this reasoning there must be an injective mapping of the sorts in specification B2 to those in specification C and there does not exist s_1, $s_2 \in$ *Sorts*(B2) such that $s_1 \neq s_2$ and $f'_S(s_1) = f'_S(s_2)$. Similar reasoning applies to there being an injective mapping of operations from B2 to specification C. The sorts s_1 and s_2 and operations f_1 and f_2 as described above cannot exist. By this contradiction the pushout of an extension morphism along any other specification morphism must result in an extension morphism.

Conservative Extensions: For purposes of contradiction, given a conservative extension morphism f:A→B1 and morphism g:A→B2, assume that the pushout of f along g, f':B2→D, is not a conservative extension morphism. By the proof above it is known that f' is an extension morphism so it must not be the case that *Axioms*(C) $\vdash_C \beta$ for all $\beta \in \Sigma 2$-sentences iff *Axioms*($\Sigma 2$) $\vdash_{\Sigma 2} \beta$ where $\Sigma 2$ represents both $B2^{f'}$, the translated source specification of f', as well as the conservative translation of specification B2 in our pushout construction method. Thus, since specification C combines the (translated) axioms of $\Sigma 1$ and $\Sigma 2$, $\Sigma 1$ must contain axioms that affect whether *Axioms*(C) $\vdash_C \beta$ for some $\Sigma 2$-sentence β. However, since f is a conservative extension morphism we know that *Axioms*(B1) $\vdash_{B1} \alpha$ for all $\alpha \in$ A-sentences iff *Axioms*(A^f) $\vdash_A \alpha$ where A^f is the translated source specification of f. Since $\Sigma 1 \cong B1$, $\Sigma 1$ is also a conservative extension of A and the same properties apply for axioms over the translated signature of specification A. When the axioms from specifications $\Sigma 1$ and $\Sigma 2$ are combined in specification C they share references to sorts and operations that originate in specification A. Specifically, the axioms in $\Sigma 1$ that make use of the sorts and operations that originate in specification A *do not* modify the entailment relation over the translated A-sentences and therefore do not modify the entailment relation over that same shared signature coming from specification $\Sigma 2$ (B2). Those axioms *cannot* modify the entailment relation for any sentence that involves sorts or operations that do not originate in specification A. The axioms in $\Sigma 1$ that make use of the extended sorts and operations (those that do not originate in specification A) *cannot* modify the entailment relation over any $\Sigma 2$ sentence. Thus the axioms of $\Sigma 1$ either do not modify the entailment relation over the signature of $\Sigma 2$ because they are conservative in nature or they cannot modify the entailment relation because they involve signature elements that are not shared between the specifications. By this contradiction we know that specification $\Sigma 1$ contains no axioms that affect whether *Axioms*(C) $\vdash_C \beta$ for some $\Sigma 2$-sentence β and our original assumption is wrong and *f'* is a conservative extension.

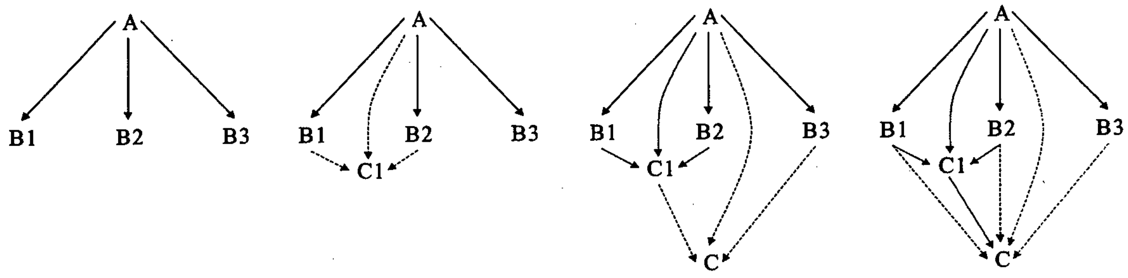
<u>Definitional Extensions</u>: For purposes of contradiction, given a definitional extension morphism *f*:A→B1 and morphism *g*:A→B2, assume that the pushout of *f* along *g*, *f '*:B2→D, is not a definitional extension morphism. By the proof above it is know that *f '* is a conservative extension morphism, so there must exist a sort in specification C that is not covered by the morphism *f '*:B2→D and that is not a constructed sort, or there must exist an operation in specification C that is not covered by the morphism *f '*:B2→D that is not total and functional. However, the sorts and operations that are not covered by the morphism from B2 originate from $\Sigma 1$ (B1). Since *f* :A→B1 is a d-morphism, these sorts are all constructed sorts and the operations are all total and functional. By construction, the translated axioms of specification B1 ($\Sigma 1$), even those that are implicit in the definition clauses and sort-axioms, are part of specification C. Because the entailment relation is monotonic and consistent under translation, the properties for the unmapped sorts and operations remain unchanged, i.e. all extended sorts are constructed sorts and all extended operations are total and functional because these properties are still theorems of the translated axioms. By this contradiction we know our original assumption to be wrong and *f'* is a definitional extension. □

C.2.3 Colimit

In the category *Spec*, a pushout is used to merge two specifications based on a third specification that indicates how the other two specifications relate to each other. A colimit in *Spec* is used to construct the minimum aggregate specification for an arbitrary diagram of component specifications. While a pushout is the minimal object that completes a commuting square, a colimit is the minimal object of an arbitrary diagram of objects and arrows that form a cocone (a collection of arrows with a common codomain) in a commuting diagram. A pushout is a particular example of a colimit.

Category *Spec* has all pushouts for any pair of morphisms of the form B1←A→B2 based on the construction method presented in Definition A.15. According to Goldblatt [Gol84] any category that has pushouts and an initial object has colimits for arbitrary diagrams, i.e. is cocomplete. For the Category *Spec*, the initial object is the empty specification as it has a unique morphism (the empty morphism) to any object in *Spec*. Thus the category *Spec* has colimits for all diagrams.

Figure C-9 is an example of a simple colimit and its meaning defined in terms of pushouts. The choice for the intermediate pushout (B1←A→B2, B1←A→B3, or B2←A→B3) is arbitrarily chosen in Figure C-9(b) to be B1←A→B2. The second pushout, Figure C-9(c), uses the result of the first pushout to "add in the structure" of the remaining object to form the colimit object, C. Regardless of the intermediate pushout choices, the resultant C-object will be the minimal cocone object and hence the colimit.



a. Initial diagram

b. 1st pushout

c. 2nd pushout

d. Colimit

Figure C-9. Colimit example

In the category *Spec*, given any base diagram of specifications and morphisms, the colimit can be mechanically constructed [GB84]. Component specifications, each representing a part of the overall problem being specified, can be combined via colimit to form a larger specification that represents the requirements of the problem as a whole.

Since a diagram of specifications and specification and morphisms can be depicted as a graph, a *Spec* diagram can be represented syntactically as a graph by labeling nodes and arcs with specifications and specification morphisms. The following Specware language code is used to develop the diagram for Example C.17, a set of flags. The "nodes" statement lists the specifications in the diagram (Set, One-Sort, and Flag) and the "arcs" statement identifies (defines) the morphisms between those specifications. This limited diagram statement forms the basis of the diagram statement developed in Chapter 5.

```
Diagram Set-of-Flags is
  nodes Set, One-Sort, Flag
  arcs One-Sort->Set:  {X->E},
       One-Sort->Flag: {X->Flag}
end-diagram
```

The following Specware language code is used to develop the colimit object of the Set-of-Flags diagram. Thus specification Set-of-Flags is the colimit of the diagram Set-of-Flags.

```
Spec Set-of-Flags is
  Colimit of Set-of-Flags
```

Using this syntax it is possible to specify and take the colimit of any finite diagram of specifications and specification morphisms. In the next section the various refinement mechanisms are described and the mechanism by which the refinements of the individual component specifications of a diagram can be combined to produce a refinement of the colimit of the diagram is described.

C.3 Refinement of Specifications

So what does it mean for one specification to be a refinement of another? Informally, if every possible implementation of specification B is also an implementation of specification A, then specification B is a refinement of specification A [ST88]. By Proposition B.39 if there is a specification morphism σ from specification A to specification B, then all models (implementations) of B will also be models (implementations) of A because of the σ-reduct functor. Since the σ-reduct functor need not be surjective there may be some models of specification A that cannot be reached by the σ-reduct functor from models of specification B. This means that the design choice (refinement) has eliminated some of the possible implementation models of specification A. Since the σ-reduct functor need not be injective there may be many models of specification B that are mapped to the same specification A model by the σ-reduct functor. This means that there may be many implementations (carrier sets and functions) associated with the sorts and operations that are not in the image of σ. (The refinement design choice that introduced some underlying implementation sorts and operations has not constrained them so that a single class of carrier sets and functions can implement them.)

Using a morphism as the refinement relation between specifications tightly couples the two specifications in that all of the sorts and operations of the source specification must have a direct map in the target specification. An example morphism refinement is Abstract-Set → Concrete-Set where the spec Concrete-Set may contain additional "implementation" sorts and operations and/or may contain constructive definitions for the Abstract-Set sorts and operations. In practical terms, a morphism between specifications means that if one can implement the target specification of the morphism then one has also developed an implementation of the source specification.

One issue with using a morphism as the refinement mechanism is that it does not expose any of the internal structure of the target specification. Not only must one refine (implement) the codomain specification sorts and operations that are targets of the morphism from the source specification, but one must also implement the additional supporting sorts and operations that are in the codomain specification. While hiding structure may be good from an abstraction standpoint, it is bad if one is trying to specify design information. If one assumes that the target of the specification morphism is an extension of some other specification, then this internal design structure is exposed and can be manipulated.

C.3.1 Interpretations

Proposition B.39 indicates that a specification morphism can be used as a refinement mechanism. However, a refinement of a specification by morphisms is not powerful enough to represent the types of refinement that occur in software-engineering as it is rare that an existing specification (or implementation) can serve directly as the target of a morphism. Instead, an existing specification must usually be extended in order to provide the appropriate sorts and operations that serve as a target for a specification morphism.

This more general refinement relation between the source specification and the extension of a target specification is called an interpretation [Ehr82]. In an interpretation, there is a morphism from the source specification to a definitional extension of the target specification. A definitional extension morphism is used because this morphism can be viewed as an implementation construction mechanism as discussed in Section C.1.5.

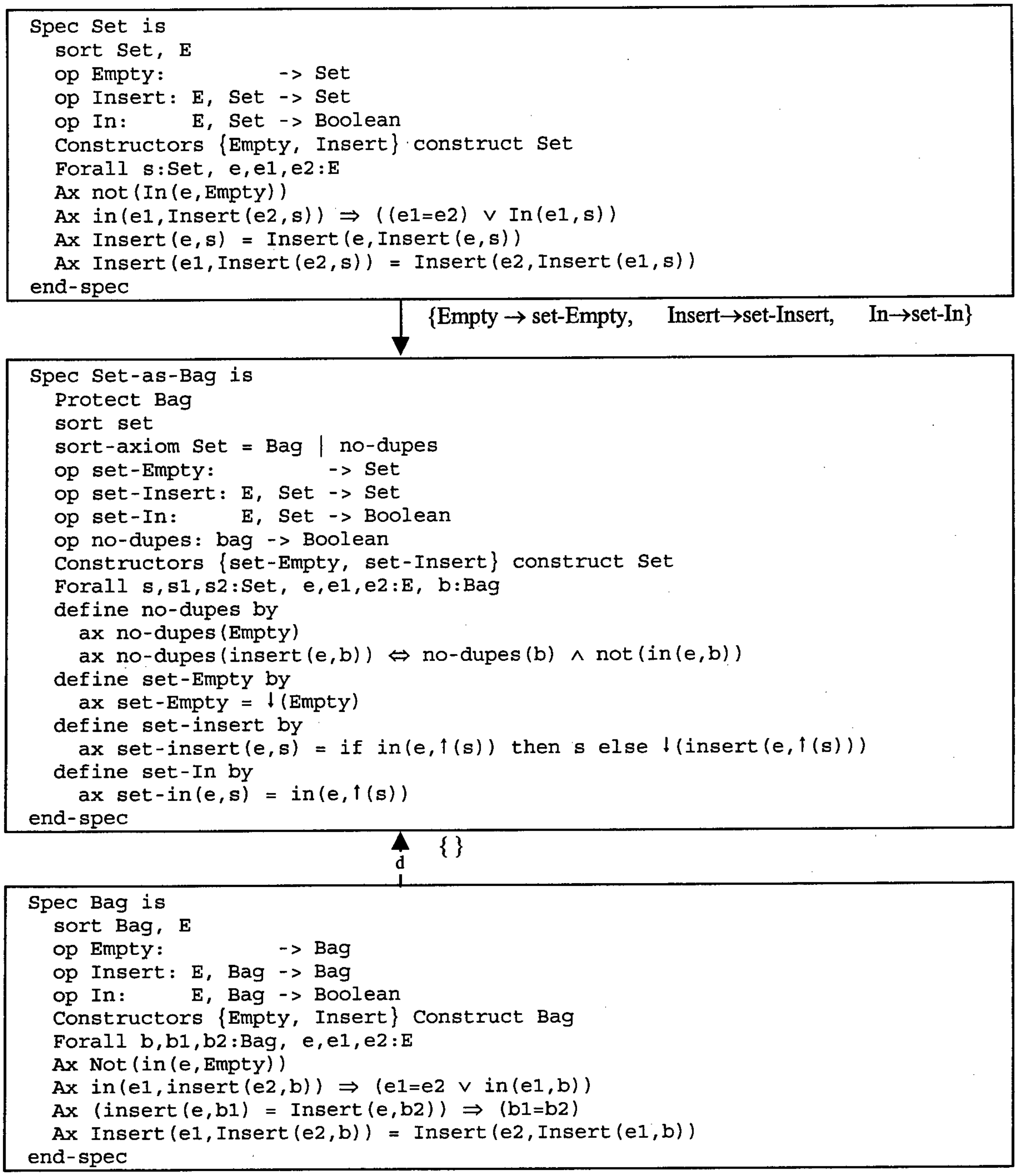
Definition C.20. An *interpretation*, $i = \langle f, g \rangle$, consists of a specification morphism f and a definitional extension morphism g with a common codomain specification, cod(f) = cod(g). The domain of the specification morphism is called the domain (or source) of the interpretation, dom(i) = dom(f). The domain of the definitional extension morphism is called the codomain (or target) of the interpretation, cod(i) = dom (g) and the codomain of both morphisms is called the mediator of the interpretation, mediator(i) = cod(f) = cod(g). An interpretation from specification A to specification B is depicted as

$$A \rightarrow A\text{-as-}B \xleftarrow{d} B, \qquad A \rightarrow B^{+} \xleftarrow{d} B, \text{ or} \qquad A \Rightarrow B.$$

Thus an interpretation from specification A to specification B is a morphism from specification A to a specification B^{+} that is a definitional extension of specification B that has added (fully defined) A-like sorts and operations. Specification A is interpreted as (refined to, implemented as, etc) a definitional extension of specification B. The morphism refinement generalizes to an interpretation refinement, as any specification morphism $A \rightarrow B$ can be generalized to an interpretation, i.e. $A \rightarrow B \xleftarrow{id} B$.

Example C.21. An interpretation from specification Set to specification Bag, Set ⇒ Bag, via the mediator Set-as-Bag, is depicted in Figure C-10. The extensions in the mediator Set-as-Bag are set-like sorts and operations that are defined using the sorts and operations of specification Bag. This interpretation means that any implementation of Bag may be extended with set-like functions that is an implementation of both the Set-as-Bag mediator specification and by Proposition 3.8, the Set specification.

Notation: The symbols ↑ and ↓ in specification Set-as-Bag in Figure C-10 are implicit operations that are defined based on the subsort relation in that specification, i.e. the text "`sort-axiom Set = Bag | no-dupes`". Up-arrow, ↑, is the super-sort operation and, for the given example, is defined as follows: ↑ = (relax no-dupes) where ↑:Set→Bag takes in a subsort and returns the sort from which it was constructed. Down-arrow, ↓, is the inverse of ↑ but is only a partial function over its domain. Down-arrow is defined as follows: ↑(↓(s)) = s and ↑ is undefined outside the range of ↓. It is partially defined as only the "bags" that have no duplicates can be directly "converted" to the set subsort. The obvious proof obligation associated with the use of ↓ is that it only be used in those contexts where its restriction (no-dupe(s)) ensures it is defined.



```
Spec Set is
  sort Set, E
  op Empty:          -> Set
  op Insert: E, Set -> Set
  op In:      E, Set -> Boolean
  Constructors {Empty, Insert} construct Set
  Forall s:Set, e,e1,e2:E
  Ax not(In(e,Empty))
  Ax in(e1,Insert(e2,s)) ⇒ ((e1=e2) ∨ In(e1,s))
  Ax Insert(e,s) = Insert(e,Insert(e,s))
  Ax Insert(e1,Insert(e2,s)) = Insert(e2,Insert(e1,s))
end-spec
```

```
Spec Set-as-Bag is
  Protect Bag
  sort set
  sort-axiom Set = Bag | no-dupes
  op set-Empty:          -> Set
  op set-Insert: E, Set -> Set
  op set-In:      E, Set -> Boolean
  op no-dupes: bag -> Boolean
  Constructors {set-Empty, set-Insert} construct Set
  Forall s,s1,s2:Set, e,e1,e2:E, b:Bag
  define no-dupes by
    ax no-dupes(Empty)
    ax no-dupes(insert(e,b)) ⇔ no-dupes(b) ∧ not(in(e,b))
  define set-Empty by
    ax set-Empty = ↓(Empty)
  define set-insert by
    ax set-insert(e,s) = if in(e,↑(s)) then s else ↓(insert(e,↑(s)))
  define set-In by
    ax set-in(e,s) = in(e,↑(s))
end-spec
```

```
Spec Bag is
  sort Bag, E
  op Empty:          -> Bag
  op Insert: E, Bag -> Bag
  op In:      E, Bag -> Boolean
  Constructors {Empty, Insert} Construct Bag
  Forall b,b1,b2:Bag, e,e1,e2:E
  Ax Not(in(e,Empty))
  Ax in(e1,insert(e2,b)) ⇒ (e1=e2 ∨ in(e1,b))
  Ax (insert(e,b1) = Insert(e,b2)) ⇒ (b1=b2)
  Ax Insert(e1,Insert(e2,b)) = Insert(e2,Insert(e1,b))
end-spec
```

Figure C-10. Interpretation: Set as Bag

An interpretation such as that in Figure C-10 can be represented syntactically by the following Specware language code:

```
Interpretation Set => Bag is
  Mediator Set-as-Bag
  Dom-to-med {Empty → set-Empty, Insert→set-Insert, In→set-In}
  Cod-to-med {}
```

The interpretation in Figure C-11 is depicted in two dimensions so that the meaning of an interpretation can be better related to software-engineering concepts. Specification A is thought of as being on the abstract requirement level and specification B is thought of as being on the concrete implementation level even though all the specifications (levels) may be very abstract. The vertical arrow represents a design decision and a change in layers that has (probably) eliminated some of the possible implementations. (Some of the models of specification A are no longer represented amongst the models of specification B^+ because the design decision represented by the vertical arrow has eliminated them.) The horizontal arrow also represents a design decision; it details how an existing implementation (models of B) can be extended to provide an implementation of specification A. The horizontal arrow is a d-morphism that represents functionality that has been added to (and constructed from) an existing class of models (implementations). The angled double arrow represents the interpretation that is composed of the layer changing and model reducing vertical morphism combined with the functionality adding and model extending horizontal morphism.

$$\begin{tikzcd}
\text{Abstract Requirement} & A \arrow[d] \arrow[dr, Rightarrow] & \\
\text{Concrete Implementation} & B^{+} & B \arrow[l, "d"]
\end{tikzcd}$$

Figure C-11. Interpretations and refinement layers

Any implementation (model) of the target specification of an interpretation, specification B in Figure C-11, can be extended (sometimes mechanically) to implement the mediator specification and hence the source specification, specification A in Figure C-11. More formally, there is a functor that takes models of specification B to models of specification A. Thus in one sense, all models of specification B are models of specification A, which fits the requirement for a refinement [ST88]. In Section C.1.5 the functor that uniquely extends models of a specification to models of its definitional extension (specification B to models of specification B^+ in Figure C-11) was described. A reduct functor takes models of specification B^+ to models of specification A. By composing the two functors, a functor that takes models of specification B to models of specification A is derived, thus justifying the use of interpretations as a refinement mechanism.

If one did not already have a concrete implementation of the target specification of the interpretation, specification B, then one could use the interpretation refinement mechanism a second time to refine specification B by itself. Thus in one sense the problem has been broken down and partitioned better than with the morphism refinement mechanism. With an interpretation refinement from specification A to specification B, to implement specification A one must often only implement specification B as the definitional extension morphism from B to B^+ may be used to mechanically extend models of B to be models of B^+ and hence models of A.

C.3.2 Interpretations and definitional extensions

Given a morphism $A \rightarrow C$ and a refinement (interpretation) of specification A, $A \Rightarrow B$, it would be useful if a compatible refinement of specification C could be mechanically generated from the interpretation $A \Rightarrow B$. Unfortunately such a construction is mechanical only if the morphism $A \rightarrow C$ is a definitional extension.

Proposition C.22. Given an interpretation $A \Rightarrow B$ and a definitional extension morphism $A \xrightarrow{d} C$, an interpretation $C \Rightarrow B$ can be mechanically generated.

Proof: On the left in Figure C-12 is a diagram consisting of an interpretation $A \Rightarrow B$ and a definitional extension morphism $A \xrightarrow{d} C$. The pushout of (sub)diagram $C \xleftarrow{d} A \rightarrow B^+$ results in a specification B^{++} and morphisms $C \rightarrow B^{++}$ and $B^+ \xrightarrow{d} B^{++}$ as depicted on the right in Figure C-12. By composing d-morphisms $B \xrightarrow{d} B^+$ and $B^+ \xrightarrow{d} B^{++}$, the d-morphism $B \xrightarrow{d} B^{++}$ is derived as depicted on the right in Figure C-12. The (sub)diagram $C \rightarrow B^{++} \xleftarrow{d} B$ is the interpretation $C \Rightarrow B$. □

$$\begin{tikzcd}
A \arrow[r] \arrow[d, "d"] & B^+ & B \arrow[l, "d"'] \\
C & &
\end{tikzcd}
\qquad
\begin{tikzcd}
A \arrow[r] \arrow[d, "d"] & B^+ \arrow[d, dashed, "d"] & B \arrow[l, "d"'] \arrow[dl, dashed, "d"] \\
C \arrow[r, dashed] & B^{++} &
\end{tikzcd}$$

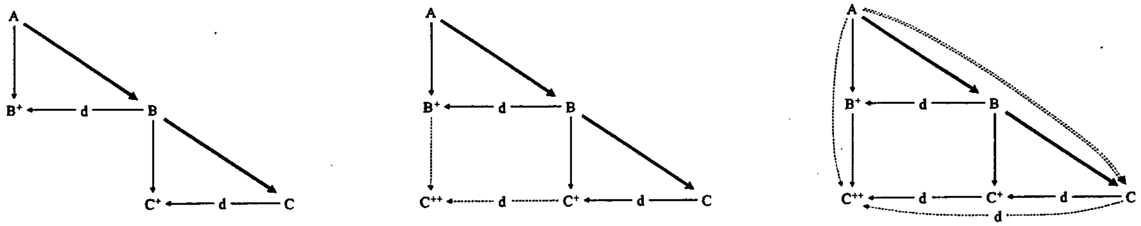
Figure C-12. Constructing an interpretation for a definitional extension

The automatic construction of the interpretation $C \Rightarrow B$ has mechanically taken the definitional extension axioms in $A \xrightarrow{d} C$ and transferred and translated them so that they are definitional extension axioms of specification B^+. From the perspective of models this makes sense; if a model of specification A is uniquely extended then an implementation of that model should be able to be uniquely extended in the same manner.

C.3.3 Composition of interpretations

For interpretations to be a viable means for refining specifications, successive interpretations should compose. If interpretations compose, then the design decisions made in each individual intermediate interpretation can be combined into a single interpretation reflecting all of the design decisions. This would enable the initial requirement specification to be viewed as a definitional extension of the final design specification.

Definition C.23. Arbitrary interpretations $A \Rightarrow B$ and $B \Rightarrow C$ that have a common codomain/domain specification compose to form an interpretation $A \Rightarrow C$ as depicted in Figure C-13. Figure C-13 (a) depicts two interpretations, $A \rightarrow B^+ \xleftarrow{d} B$ and $B \rightarrow C^+ \xleftarrow{d} C$ with a common domain/codomain. The A layer is implemented by (refined into) the B layer that in turn is implemented by (refined into) the C layer. In Figure C-13(b), a pushout from diagram $B^+ \xleftarrow{d} B \rightarrow C^+$ is used to mechanically construct specification C^{++}. Finally, in Figure C-13(c) the vertical morphisms and horizontal d-morphisms are composed to form the interpretation $A \rightarrow C^{++} \xleftarrow{d} C$.



a. $A \Rightarrow B \Rightarrow C$ Two Interpretations

b. $B^+ \leftarrow B \rightarrow C^+$ Pushout Construction of C^{++} Mediator

c. $A \Rightarrow C$ Interpretation by Morphism Composition

Figure C-13. Composition of interpretations in *Spec*

For example, in Figure C-14 there is an interpretation from specification Bag to specification List, $Bag \Rightarrow List$. This can be composed with the interpretation $Set \Rightarrow Bag$ to form an interpretation $Set \Rightarrow List$, Figure C-15.

Notation: The Angled-Arrow, ↙, in Figure C-14 is defined to be the quotient operation based on the quotient sort. In specification List-as-Bag, Figure C-14, ↙: List→Bag, is defined as follows:

↙ = (quotient perm).

```
Spec Bag is
  sort Bag, E
  op Empty:          -> Bag
  op Insert: E, Bag -> Bag
  op In:     E, Bag -> Boolean
  Constructors {Empty, Insert} Construct Bag
  Forall b,b1,b2:Bag, e,e1,e2:E
  Ax Not(in(e,Empty))
  Ax in(e1,insert(e2,b)) ⇒ (e1=e2 ∨ in(e1,b))
  Ax (insert(e,b1) = Insert(e,b2)) ⇒ (b1=b2)
  Ax Insert(e1,Insert(e2,b)) = Insert(e2,Insert(e1,b))
end-spec
```

↓ {Empty → bat-Empty, Insert→bag-Insert, In→bag-In}

```
Spec Bag-as-List is
  Protect List
  sort Bag
  sort-axiom Bag = List/Perm
  op Bag-Empty:          -> Bag              op Perm:   List, List -> Boolean
  op Bag-Insert: E, Bag -> Bag               op Count:     E, List -> Nat
  op Bag-In:     E, Bag -> Boolean
  Constructors {bag-Empty, bag-Insert} Construct Bag
  Forall b,b1,b2:Bag, e,e1,e2:E, l,l1,l2:List
  define Count by
    ax Count(e,Empty) = 0
    ax Count(e1,insert(e2,l)) = if e1=e2 then Count(e1,l) + 1
                                          else Count(e1,l)
  define Perm by
    ax Perm(l1,l2) ⇔ (Count(e,l1) = Count(e,l2))
  define bag-Empty
    ax bag-Empty = ↙(Empty)
  define bag-Insert
    ax bag-insert(e,↙(l)) = ↙(Append(e,l))
  define bag-In
    ax Not(bag-In(e,bag-Empty))
    ax bag-In(e1,bag-Insert(e2,l)) ⇔ (e1=e2 ∨ bag-In(e1,l))
end-spec
```

↑ d { }

```
Spec List is
  sort List, E
  op Empty:          -> List
  op Append: E, List -> List
  op In:     E, List -> Boolean
  Constructors {Empty, Append} Construct List
  Forall s,s1,s2:List, e,e1,e2,e3,e4:E
  Ax Not(in(e,Empty))
  Ax in(e1,Append(e2,s)) ⇒ (e1=e2 ∨ in(e1,s))
  Ax (Insert(e1,s1) = Insert(e2,s2)) ⇔ (e1=e2 ∧ s1=s2)
  Ax (Insert(e1,Insert(e2,s)) = Insert(e3,Insert(e4,s)) ⇔ (e1=e3 ∧ e2=e4)
end-spec
```

Figure C-14. Interpretation: Bag as List

If the source of an interpretation is a commonly used specification, the interpretation should be placed in a library so that it may be reused. The next section addresses how collections of interpretations are currently used to refine aggregate structures (diagrams).

$$\begin{tikzcd}
\text{Set} \arrow[d] \arrow[dr, thick] \arrow[ddrr, dashed, bend left] \arrow[dd, dashed, bend right] & & \\
\text{Bag}^+ \arrow[d] & \text{Bag} \arrow[l, "d"] \arrow[d] \arrow[dr, thick] & \\
\text{List}^{++} & \text{List}^+ \arrow[l, "d"] & \text{List} \arrow[l, "d"] \arrow[ll, dashed, bend left, "d"]
\end{tikzcd}$$

Figure C-15. Interpretation: Set as List

C.4 Refinement of Diagrams of Specifications

Colimits are used to form an aggregate specification from component specifications. Diagram refinement is a technique that enables the interpretations from component specifications to be combined so that their aggregate forms an interpretation of the aggregate of the components. In order to put interpretations together in a fashion similar to how the specifications are put together there needs to exist morphisms between interpretations.

C.4.1 Interpretation morphisms

An interpretation morphism (morphism between interpretations) ensures that the "structure" of one interpretation is preserved in another in the same manner that a specification morphism ensures that the "structure" of one specification is preserved in another. An interpretation morphism consists of specification morphisms between the domain, mediator, and codomain specifications of the interpretations.

Definition C.24. An *interpretation morphism* is a 5 tuple, *im* = <A ⇒ B, C ⇒ D, f:A→C, g:B^+→D^+, h:B→D>, where dom(*im*) = A⇒B, cod(*im*) = C⇒D, dom-morphism(*im*) = f, cod-morphism(*im*) = h, med-morphism(*im*) = g, consisting of the domain and codomain interpretations and three morphisms between the domain, codomain and mediator specifications of the domain interpretation to the domain, codomain and mediator specifications of the codomain interpretations, such that the diagram in Figure C-16 commutes.

$$\begin{tikzcd}
\text{Domain Interpretation} \arrow[d, Rrightarrow, "\text{Interpretation Morphism}"] & A \arrow[r] \arrow[d] & B^{+} \arrow[d] & B \arrow[l, "d"'] \arrow[d] \\
\text{Codomain Interpretation} & C \arrow[r] & D^{+} & D \arrow[l, "d"]
\end{tikzcd}$$

Figure C-16. Interpretation Morphism I

The commutativity of the specifications making up the interpretation morphism diagram, Figure C-16 , is evidence that the structure of one interpretation is preserved in the other. The full structure is often left off a diagram involving interpretations and interpretation morphisms in order to reduce the clutter. Figure C-17 is an alternate depiction of the interpretation morphism of Figure C-16.

$$\begin{tikzcd}
A \arrow[r, Rightarrow, ""{name=U, below}] & B \\
C \arrow[r, Rightarrow, ""{name=L, above}] & D
\arrow[from=U, to=L, Rrightarrow]
\end{tikzcd}$$

Figure C-17. Interpretation Morphism II

C.4.2 The category *Interp*

In order to construct an aggregate refinement from a collection of related refinements the refinements themselves must be treat as objects that can be combined. Interpretations and interpretation morphisms can be treated as the objects and arrows of a category, and the categorical colimit and composition constructs can be used to construct such a refinement aggregate.

Proposition C.25. Interpretations and Interpretation morphisms form a category, *Interp,* where the objects are interpretations between specifications and the arrows are interpretation morphisms.

Proof: For any given interpretation $A \rightarrow B^{+} \leftarrow d - B$, the identity interpretation morphism is the 5-tuple $<A \rightarrow B^{+} \leftarrow d - B, A \rightarrow B^{+} \leftarrow d - B, A_{id}, B^{+}_{id}, B_{id}>$, where the domain, codomain and mediator specification morphisms are the identity morphisms for those specifications. Interpretation morphisms compose because the individual specification morphisms making up the interpretation morphisms compose. Composition of Interpretation morphisms is associative because composition of the individual specification morphisms making up the interpretation morphism is associative. □

There are three obvious forgetful functors (functors that remove underlying object and arrow structure) from *Interp* to *Spec* based on the domain, codomain and mediator specifications making up an interpretation and the morphisms between those specifications that make up an interpretation morphism.

Proposition C.26. The functor Dom:*Interp*→*Spec* is the pair of functions <$\mathrm{Dom}_{\mathrm{Object}}$, $\mathrm{Dom}_{\mathrm{Arrow}}$> where $\mathrm{Dom}_{\mathrm{Object}}(i)$ = dom(i) for all $i \in$ Objects(*Interp*) and $\mathrm{Dom}_{\mathrm{Arrow}}(im)$ = dom-morphism(im) for all $im \in$ Arrows(*Interp*). The functors Cod: *Interp*→*Spec* and Med *Interp*→*Spec* are similarly defined over the functions cod and cod-morphism, and the functions mediator and med-morphism respectively.

Proof: The identity arrows are preserved as $\mathrm{Dom}_{\mathrm{Arrow}}(\mathrm{id}_{im}) = \mathrm{id}_{\mathrm{Dom}_{\mathrm{Object}}(im)}$ for all $im \in$ Objects(*Interp*). Composite arrows are preserved as $\mathrm{Dom}_{\mathrm{Arrow}}(g \circ f) = \mathrm{Dom}_{\mathrm{Arrow}}(g) \circ \mathrm{Dom}_{\mathrm{Arrow}}(f)$ whenever $g \circ f$ is defined in *Interp*. □

A diagram in the category *Interp* consists of a collection of nodes labeled with interpretations and a collection of arcs between the nodes labeled with interpretation morphisms. The functors Dom, Cod, and Med reduce *Interp* diagrams to *Spec* diagrams. Thus for an *Interp* diagram D, the notation Dom(D) (and Cod(D) and Med(D)) is used to reference the diagrams in Spec that have the same shape of diagram D but that consist of the domain (and codomain and mediator) specifications and morphisms of the interpretations in the interpretation diagram.

C.4.3 Colimits of interpretations

Since an interpretation is constructed from specifications and specification morphisms and an interpretation morphism is constructed from specification morphisms, it is natural that a colimit in the category *Interp* be constructed from the colimits of its components in the category *Spec*. This colimit operation is what is used to form an aggregate interpretation from a diagram of interpretations.

Proposition C.27. A colimit object C of a diagram D in the category *Interp* can be constructed by taking the colimits of diagrams Dom(D), Cod(D), and Med(D) to construct the domain, codomain, and mediator objects of C and then constructing witness arrows from specifications dom(C) and cod(C) to specification mediator(C). The interpretation morphisms to this constructed interpretation in the category *Interp* are the cocone morphisms from the three colimit morphisms of the Dom, Cod, and Med diagrams in *Spec*.

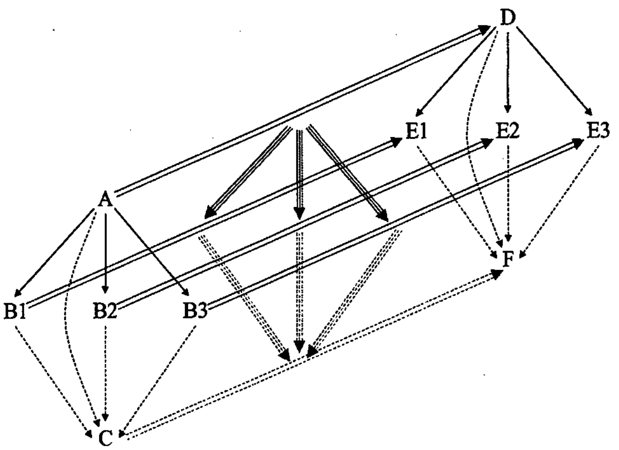
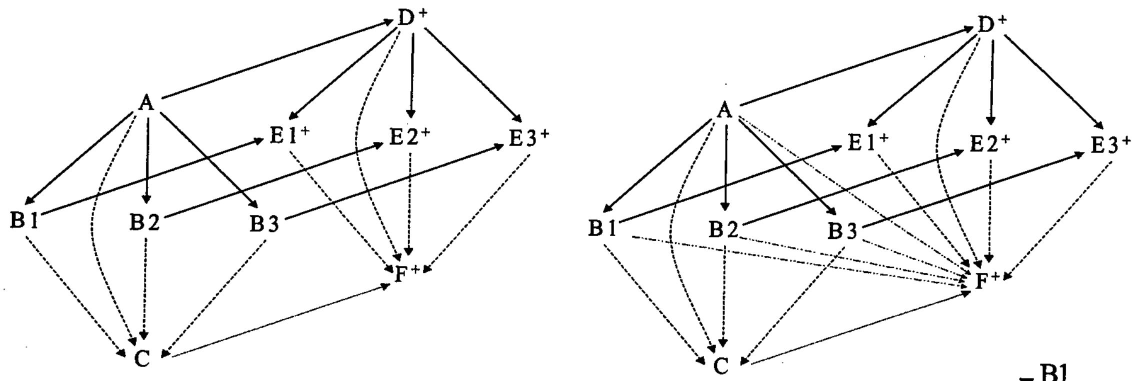


Figure C-18. Colimit in category *Interp*

Proof: In order to prove the construction above is valid it must proven that it results in a commuting diagram where the constructed interpretation has a unique interpretation morphism to the cocone object (interpretation) of any other commuting diagram in category *Interp*. Figure C-18 represents a typical example of an interpretation diagram. The collections of objects and solid arrows in the top half of the two diagrams in Figure C-18 represent the initial *Interp* diagram, i.e. four interpretations (double lines) with three interpretation morphisms connecting them (triple lines). The collections of dashed arrows in the bottom half of Figure C-18 represent the constructed colimit object and cocone morphisms.

Existence of Colimit Object: It is easy to see how the domain, codomain and mediator objects of the desired colimit (interpretation) object are constructed by taking colimits in category Spec over the domain, codomain and mediator (sub)diagrams of the interpretation diagram. In Figure C-19(a) the colimits for the domain and mediator specifications are depicted. There exists a unique arrow from the colimit object of the domain specifications to the colimit object of the mediator specifications (and from codomain colimit to mediator colimit) as the mediator colimit object is a cocone object of the domain (and codomain) specification diagrams. In Figure C-19(b) the cocone arrows to F^+ can be composed with the domain to mediator arrows of the interpretations so that specification F^+ is a cocone object with respect to the diagram of domain specifications. Since specification C is a colimit object and specification F^+ is a cocone object from the same diagram of specifications, there is a unique arrow from C to F. The arrow from the codomain specification to the mediator specification can be similarly constructed. The codomain to mediator arrow is guaranteed to be a definitional extension morphism as d-morphisms are closed under parallel composition [SJ95].



a. C and F$^+$ are colimit objects

b. F$^+$ is a cocone object of diagram: $$\begin{tikzcd} & B1 \\ A \arrow[ur] \arrow[r] \arrow[dr] & B2 \\ & B3 \end{tikzcd}$$

Figure C-19. Constructing a witness arrow

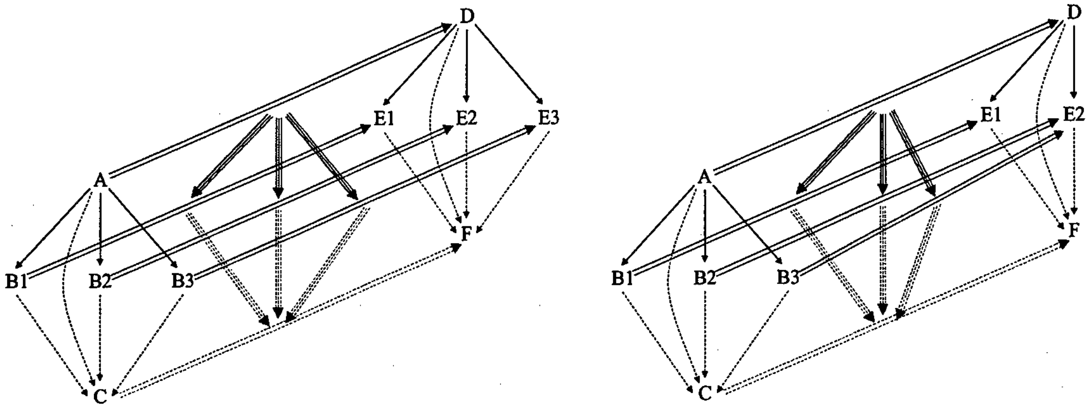
<u>Existence and Uniqueness of Arrow</u>: Assume there exists another cocone interpretation, G, from diagram D. dom(G) is a cocone object of Dom(D) and by construction dom(C) is the colimit object of Dom(D). Thus by Definition A.17, there is a unique arrow from dom(C) to dom(G). Similarly there are unique arrows from med(C) and cod(C) to med(G) and cod(G) respectively. Together these three arrows form a unique interpretation morphism from interpretation C to interpretation D. The overall diagram commutes as the three individual colimit to cocone diagrams commute. □

The interpretation colimit object for any interpretation diagram can be constructed mechanically. First take the colimits of the source and target diagrams. Every specification and morphism except the mediator specification of the interpretation object and all of the specification morphisms to the mediator specification, i.e. specification F$^+$ and morphisms to specification F$^+$ in Figure C-19 are now known. The mediator is mechanically constructed by taking the colimit of the entire diagram, including the newly constructed domain and codomain specifications. All of the missing morphisms are the mechanically constructed cocone morphisms.

C.4.4 Diagram refinement

In diagram refinement a collection of source specifications is individually refined to a collection of target specifications via interpretations. The source specifications are related in that there are specification morphisms between them. Likewise the target specifications are related by specification morphisms.

Furthermore, the interpretations have interpretation morphisms between them. If all of these specification morphisms and interpretation morphisms are consistent (the top half of the diagrams in Figure C-20 commute) then an interpretation from the colimit of the source specifications to the colimit of the target specifications (the bottom half of diagrams in Figure C-20) can be mechanically constructed.



a. Source and Target diagrams have the same shape, their consistency is obvious

b. Source and Target diagram have different but still consistent shapes

Figure C-20. Diagram refinement

The following definition for diagram refinement comes from [SJ95].

Definition C.28. Given two diagrams of specifications $D_1:G_1 \rightarrow Spec$, and $D_2:G_2 \rightarrow Spec$ where D_1 and D_2 are shape mappings from the category of graphs (unlabeled, directed multi-graphs and morphisms between graphs) to the category of specifications, and G_1 and G_2 are particular shapes (graphs), a *diagram refinement* is a pair $<\delta, \sigma>:D_1 \rightarrow D_2$ where $\delta:G_1 \rightarrow Interp$ is a diagram of interpretations and $\sigma:G_1 \rightarrow G_2$ is functor between the two shapes such that the following diagram commutes (Dom and Cod are the functors from *Interp* to *Spec*).

$$\begin{tikzcd}
 & G_1 \arrow[rr, "\sigma"] \arrow[dl, "D_1"] \arrow[d, "\delta"] \arrow[dr] & & G_2 \arrow[dl, "D_2"] \\
Spec & Interp \arrow[l, "\text{Dom}"'] \arrow[r, "\text{Cod}"] & Spec &
\end{tikzcd}$$

Figure C-21. Necessary relationship between specifications and shapes in a diagram refinement

If the source, target, and interpretation shapes are the same, as in Figure C-20(a) where $G_1=G_2$, then the interpretation between the colimits is the colimit of the diagram of interpretations. If the source and target shapes are different, as in Figure C-20(b) where $G_1 \neq G_2$, an additional step is needed. First take the colimit of the diagram of interpretations as before, then compose that interpretation with a morphism from its codomain to the colimit object of a diagram of target specifications with shape G_2, see Figure C-22.

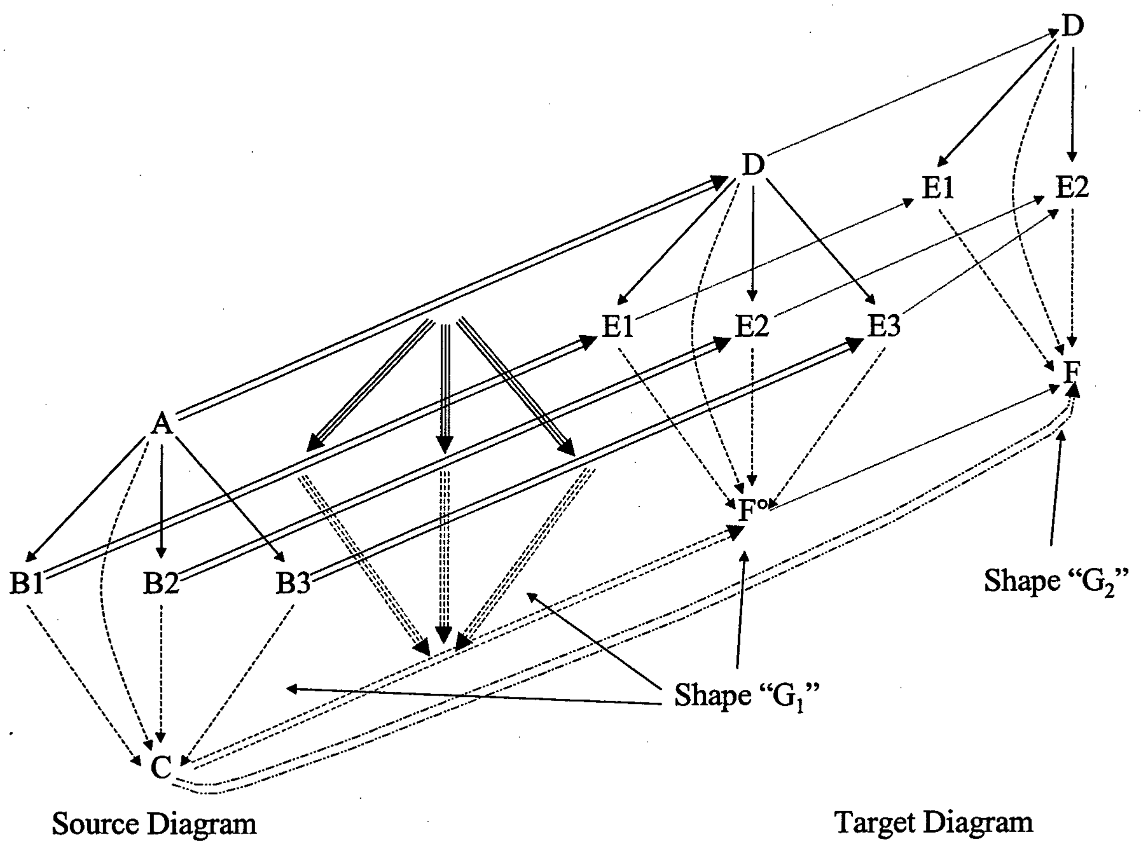


Figure C-22. Diagram refinement with shape mapping

The shape morphisms from the target diagram with shape G_1 to shape G_2 exist as the (sub)diagram that relates those shapes in Figure C-22 commutes. The morphism from specification F° to specification F exists because F is also a cocone object of the target diagram with shape G_1, see Figure C-19. The proof that interpretation C⇒F° and morphism F°→F compose to interpretation C⇒F is depicted in Figure C-23 where advantage is taken of the fact that definitional extension morphisms are preserved by pushouts.

$$\begin{tikzcd}
C \arrow[r] \arrow[dr, dashed] & F^{+} \arrow[d, dashed] & d \arrow[l] \arrow[r, dash] & F^{\circ} \arrow[d] \\
 & F^{++} & d \arrow[l, dashed] \arrow[r, dash, dashed] & F
\end{tikzcd}$$

Figure C-23. An interpretation composed with a (target) morphism

Example C.29. Refinement of Set-of-Flag to List-of-Flag. In Example C.17 (Figure C-7), The Set-of-Flags specification is developed from a pushout involving the specifications Set, One-Sort, and Flag. In Figure C-15 the Set specification is refined to the List specification via the interpretation Set ⇒ List. The assumption in this example is that the Set ADT is to be implemented as a List ADT while leaving the implementation of specification Flag alone. Thus at the start of the refinement the diagram is as depicted on the left in Figure C-24. In order to form an interpretation from specification Set-of-Flag to specification List-of-Flag a family of compatible interpretations is needed from the Set-of-Flag diagram to the List-of Flag diagram as depicted using solid arrows in the diagram on the right in Figure C-24.

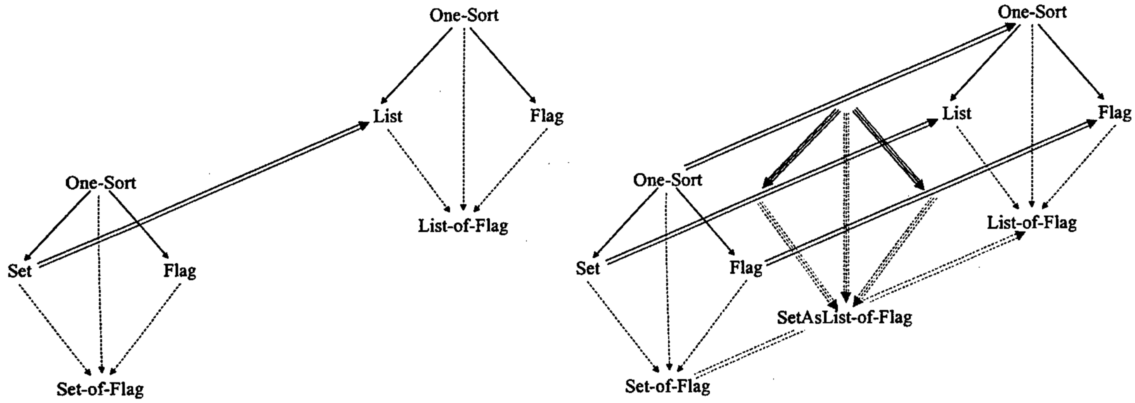


Figure C-24. Example refinement of an aggregate object

In this case the interpretation from specification One-Sort in the Set diagram to specification One-Sort in the List diagram is simply the identity interpretation, One-Sort –id→ One-Sort ←id– One-Sort. The interpretation for the Flag specifications is also the identity interpretation. To show that all three interpretations are compatible there must be interpretation morphisms (solid triple arrows) between the three interpretations. These can sometimes be mechanically inferred given the rest of the diagram [SLM98]. Once all of the solid arrows in Figure C-24 have been established, the dashed arrows can be mechanically generated by using colimit (in this case pushout) constructions and by composing various morphisms. The Specware system does not have a syntax for representing diagram refinement although one can interactively (graphically) build a diagram refinement in the Specware environment and can simulate a diagram refinement by using the lower level diagram and colimit statements.

As this small example demonstrates, it is possible to construct a larger specification from smaller component specifications and then to combine the refinements of the smaller specification into a refinement of the larger specification. This example deliberately avoided (glossed over) some of the messier details of the diagram refinement that can be found in [SLM98] and that are taken to task in Section 2.5.

C.4.5 Refining complex structure

The requirement specification for a complex application may have a great deal of "structure" as it could be recursively created using colimits and extensions of many existing specifications. For example a requirement specification may be constructed as a definitional extension of the colimit specification of a diagram of specifications where the specifications in the diagram also may contain similar "structure". In order to refine such complex structure, refinement by definitional extension and refinement by diagram refinement should compose such that refinements are rippled down the structure of the specification.

The two types of refinement do compose as diagram refinement results in a refinement of the colimit specification of that diagram, any definitional extension of that colimit specification can have its refinement mechanically generated. Thus as long as a specification is structured using colimits and definitional extensions, compatible individual refinements of the component specifications can be composed to form a refinement of the aggregate specification as a whole.

C.4.6 Applying structured refinements

As the refinements Set⇒Bag and Bag⇒List indicate, the individual refinements of specifications may have "structure" as well, where successive design decisions are made by successive interpretations. This can be extended to the diagram level where the design decisions at the larger diagram level result in the need for successive diagram refinements.

Successive diagram refinements can be composed because interpretations compose and the shape mappings compose successive diagram refinements compose as well [SJ95]. This can be accomplished by first composing the interpretations that make up the successive diagram refinements and then performing a single diagram refinement, or it can be accomplished by performing the diagram refinements individually and then composing the (successive) interpretations that result.

C.5 Code Generation

As described in Section B.2, one meaning of a specification is the class of models that satisfy the specification. A more practical meaning, however, is the class of programs in a programming language that are models of the specification. The meaning of a refinement can be defined in terms of the delta between the meanings associated with the source and target specifications, i.e. the delta between the models associated with the source specification and the models associated with the target specification. However, the goal of a transformation system is to refine a specification until it can be translated directly to code. So another meaning of a refinement is the delta that occur in the translations of the source and target specifications to code.

C.5.1 Models semantics vs. generated code semantics

Unfortunately not all so-called design specifications can be translated directly to code, as refining a specification so that it has a single model is not the same thing as refining it so that it can be translated to code. As an example, Figure C-25 contains two different definitions of the Size operations. The second definition is arguably a refinement of the first yet both definitions have the same function as their model (assuming the usual List specification semantics).

```
spec List is
     ...
  op append: E, List →List
  op Size: List → Nat
     ...
define Size by
  ax Size(Empty(L)) = 0
  ax Size(Insert(e,L)) = 1 + Size(L)
            vs.
define Size by
  ax Size(L) = if Is-Empty(L) then 0
                else 1 + Size(Tail(L))
     ...
end-spec
```

Figure C-25. Constructive vs. non-constructive definitions

C.5.2 Constructive vs. non-constructive definitions

One of the main reasons why a specification could not be translated to code is that the definitions of various sorts and operations may not be constructive in nature ("what" is defined but "how" is not). As an example, only one of the definitions in Figure C-25 is constructive. The first definition (the definition based on the constructors Empty and Insert interestingly enough) is the non-constructive definition. The second definition (based on the "destructor" Tail and the "base test" Is-Empty) is the constructive definition as there is an obvious translation to code given the existence of operations Tail and Is-Empty.

As another example, Figure C-26 contains two equivalent definitions for the operation Fermat. The equivalency of these two definitions has only recently been proven [Wil95]. The first definition expresses the problem solved by the operation but does not express how (constructively) to arrive at an answer. A constructive (algorithmic) definition would be searching for such an answer for a very long time. However, because the answers are now known, the operation Fermat can be defined as shown in the second definition in Figure C-26. The second definition is also constructive and is easily translatable to code; however, the only indication as to what problem is being solved by the operation is the user-friendly name, Fermat, of the operation. These two definitions exemplify the difference between specifying *what* and specifying *how*. The first definition is easy to understand in terms of what problem is being solved, although how to arrive at a solution is not immediately apparent. The second definition is easy to use determine to an answer, although one is unsure if it is solving the correct problem.

```
spec Fermat is
  op Fermat: Nat → Boolean

define Fermat by
    ax Fermat(n) ⇔ ∃ x,y,z:Nat | x^n + y^n = z^n

define Fermat by
    ax Fermat(n) = if (n = 0) or (n > 2) then False
                                          else True
end-spec
```

Figure C-26. Example requirement specification for which no code can be generated

C.5.3 Translation via inter-logic morphism

Assuming that certain conditions concerning the axioms are met (such as all existentials have been eliminated through design decisions), the final design specification can be translated to code via direct translation [Lin93, Loeckx87] or via an inter-logic morphism [SJ95, Meseguer89]). The inter-logic morphism method used by the Specware system is called an entailment system morphism [WSGJ96]. For this approach to work, only a "translatable" subset of the Specware specification language can be used in the final design specification, essentially conditional equational logic, plus a select set of "base" specifications that have pre-established translations (typically corresponding to built in type and operations of the programming language).

For each implementation language that the final design could be translated to, there must exist an entailment system that describes the language in its own logic and syntax. The entailment system in the implementation language enables statements in the language to be reasoned about mechanically. An entailment system morphism is then used to map the entailment system of the subset of the specification language to the entailment system of the implementation language. This entailment system morphism enables correctness in the specification language to be preserved in the implementation language. Given that this entailment system morphism exists, the specification to code transformation is accomplished by establishing a morphism (actually an interpretation) between the final requirement specification and a program in the implementation language. The definitional extension part of this interpretation between the two logics is the generated code. (A collection of pre-established base code in the target implementation language is extended definitionally so that it implements the design specification in the source specification language.)

For any particular source specification language and target implementation language there may be a collection of additional requirements on the final design specification in order for the entailment system morphism and inter-logic interpretation to work. The Specware system requires colimit independence, colimit uniqueness, constructiveness of definitions, axiom form and semantics, and sort constructor restrictions when going to Lisp and C code [WSGJ96]. While these restrictions are burdensome, they do not apply to all specifications and definitions in the refinement process, only the final design specification that is to be mechanically translated.

Appendix D. Diagram Construction

As a motivating example for the use of parameterized diagrams, the data structure for a Petri Net [Pet77] is developed using two different versions of the diagram construction method (listing a diagram's nodes and arcs). A Petri Net data structure can be specified as consisting of the following collection of smaller data structures: a set of places, a set of transitions, a map from places to natural numbers representing the number of Petri Net tokens at any given place, and bags of input and output arcs that connect places to transitions and transitions to places. This abstract structure of a Petri Net is depicted in Figure D-1.

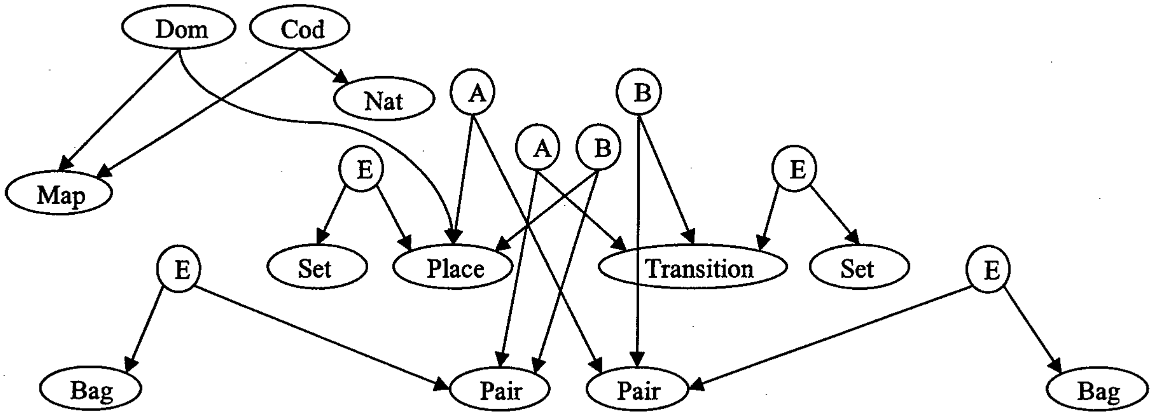


Figure D-1. Pictorial representation of the abstract data structure of a Petri Net

The Specware language code [SLM98] in Figure D-2 constructs the diagram in Figure D-1 using specifications and morphisms as the largest reusable objects. The specifier has to recreate the "fixed" portion of the diagram as well as the "variable" portion of the diagram for a number of smaller abstract data type instantiations that are built up into the larger Petri Net data type. From the code in Figure D-2 alone it is difficult to understand what is being constructed or what specification one will end up with once the colimit of the diagram is taken. Constructing such a diagram by listing its nodes and arcs is not a natural way to express such a complex data structure. Neither is drawing such a diagram using a graphical editor to place the nodes and connect them with arcs. The construction of such large, monolithic diagrams from individual specifications and morphisms is called the diagram construction approach.

```
Diagram Petri-Net is
  Nodes T1:One-Sort, T2:One-Sort, S1:Set, S2:Set, Place, Transition,
        T3:One-Sort, T4:One-Sort, P1:Pair,
        T5:One-Sort, T6:One-Sort, P2:Pair,
        T7:One-Sort, T8:One-Sort, B1:Bag, B2:Bag,
        T9:One-Sort, T0:One-Sort, Map, Empty
  Arcs
      T1 -> Place:        {X -> Place}
      T1 -> S1:           {X -> E}
      T2 -> Transition:   {X -> Transition}
      T2 -> S2:           {X -> E}
      T3 -> Place         {X -> Place}
      T3 -> P1:           {X -> A}
      T4 -> Transition:   {X -> Transition}
      T4 -> P1:           {X -> B}
      T5 -> Transition:   {X -> Transition}
      T5 -> P2:           {X -> A}
      T6 -> Place         {X -> Place}
      T6 -> P2:           {X -> B}
      T7 -> B1:           {X -> E}
      T7 -> P1:           {X -> Pair}
      T8 -> B2:           {X -> E}
      T8 -> P2:           {X -> Pair}
      T9 -> Map:          {X -> Dom}
      T9 -> Place         {X -> Place}
      T0 -> Map:          {X -> Cod}
      T0 -> Empty:        {X -> Empty.Nat}
end-diagram
```

Figure D-2. Diagram construction approach for developing a Petri Net specification

An alternative diagram construction method breaks the problem into smaller pieces, each of which performs part of the necessary construction, that build upon each other to form the overall Petri Net specification. An example of this piecemeal diagram construction approach is depicted in Figure D-3 where the Petri Net specification is developed in a sequence of steps.

The upper left diagram (and colimit) in Figure D-3 creates a diagram (and specification) that contains sets of places and sets of transitions. The upper right diagram (and colimit) in Figure D-3 creates a diagram (and specification) that has input and output arcs represented by the pairs <place, transition> and <transition, place>, respectively, by adding to the previously constructed specification. The lower left diagram (and associated specification) adds bags of input and output arcs. The final diagram (and specification) creates the data structure of a Map from Places to Nats that is used to associate a number of tokens with each place in the Petri Net data structure. Thus the specification Petri-Net5 is isomorphic to the specification that one gets when taking the colimit of the diagram developed in Figure D-2, but it was arrived at piecemeal. The overall diagram as depicted in Figure D-1 does not exist in the piecemeal approach, as in each step it is the colimit specification that is being extended and not the diagram.

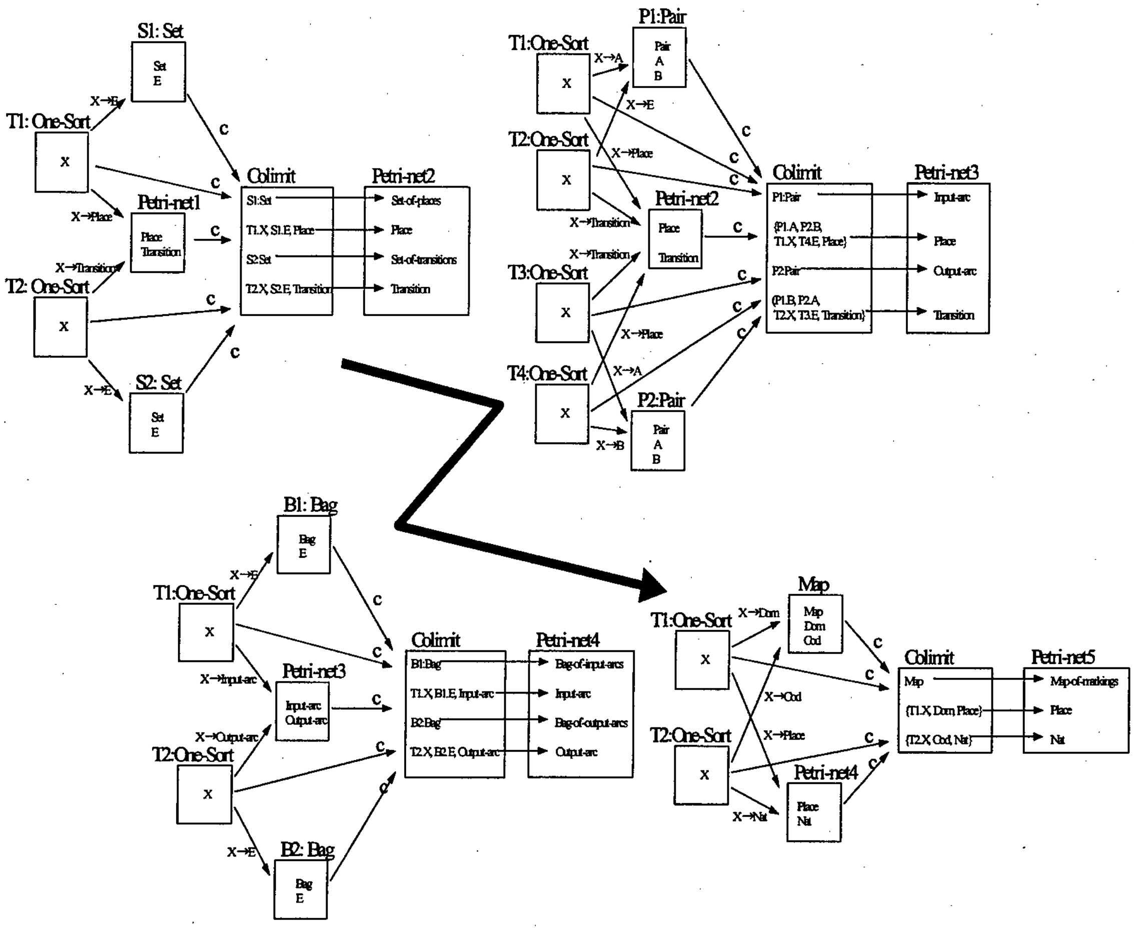


Figure D-3. Petri Net specification using four separate diagrams

The use of intermediate specifications and diagrams introduces dependencies among the specifications that are not present in the diagram construction approach used in Figure D-2. For example, in Figure D-3 and Figure D-4 it appears that one must create the diagram representations for a set of places prior to creating the diagram representations for a map from places to natural numbers, which is not the case. The order and content of the specifications Petri Net1 through Petri Net4 in Figure D-4 are somewhat arbitrary, as the piecemeal construction could be accomplished in a number of different ways. The use of the piecemeal construction constitutes a false economy, as certain kinds restructuring refinements are no longer possible based on the false dependencies induced by the use of intermediate colimit specifications.

While the construction depicted in Figure D-3 appears easier to understand from a diagrammatic standpoint, the code for it is longer than for the all-at-once diagram construction approach and the code is still very complicated, as can be seen in Figure D-4.

```
spec petri-net1 is
 translate colimit of
   diagram nodes Place:one-sort, Transition:one-sort end-diagram by
     {Place.X ->Place, Transition.X -> Transition}
end-spec
```

```
spec petri-net2 is
  colimit of diagram
      nodes T1:one-sort, S1:set, T2:one-sort, S2:set, petri-net1
      arcs
        T1 -> S1 : {X -> E},
        T1 -> petri-net1 : {X -> Place},
        T2 -> S2 : {X -> E},
        T2 -> petri-net1 : {X -> Transition}
    end-diagram
```

```
spec petri-net3 is
  colimit of diagram
      nodes T1:one-sort, T2:one-sort, P1:pair,
            T3:one-sort, T4:one-sort, P2:pair, petri-net2
      arcs
        % Input arc (Place x Transition)
        T1 -> P1 : {X -> A},
        T2 -> P1 : {X -> B},
        T1 -> petri-net2 : {X -> Place},
        T2 -> petri-net2 : {X -> Transition},
        % Output arc (Transition x Place)
        T3 -> P2 : {X -> A},
        T4 -> P2 : {X -> B},
        T3 -> petri-net2 : {X -> Transition},
        T4 -> petri-net2 : {X -> Place}
    end-diagram
```

```
spec petri-net4 is
  colimit of diagram
      nodes T1:one-sort, B1:bag, T2:one-sort, B2:bag, petri-net3
      arcs
        T1 -> B1 : {X -> E},
        T1 -> petri-net3 : {E -> p1.pair},
        T2 -> B2 : {X -> E},
        T2 -> petri-net3 : {E -> p2.pair}
    end-diagram
```

```
spec petri-net5 is
  colimit of diagram
      nodes T1:one-sort, T2:one-sort, map, petri-net4
      arcs
        T1 -> map : {X -> Dom},
        T2 -> map : {X -> Cod},
        T1 -> petri-net4 : {X -> Place},
        T2 -> petri-net4 : {X -> Nat}
    end-diagram
```

Figure D-4. Piecemeal diagram construction approach for developing a Petri Net specification

The language construct for creating and instantiating parameterized diagrams that is developed in Chapter 5 enables the specifier to reuse, parameterize and instantiate diagrams which enables a specifier to work at a higher level of abstraction.

Appendix E. Parameterized Specifications

This appendix presents examples of category-theory-based parameterization and λ-style parameterization including single parameterization, multiple parameterization, and parameterization by a diagram as well as nested and recursive instantiation.

E.1 Category theory style parameterization

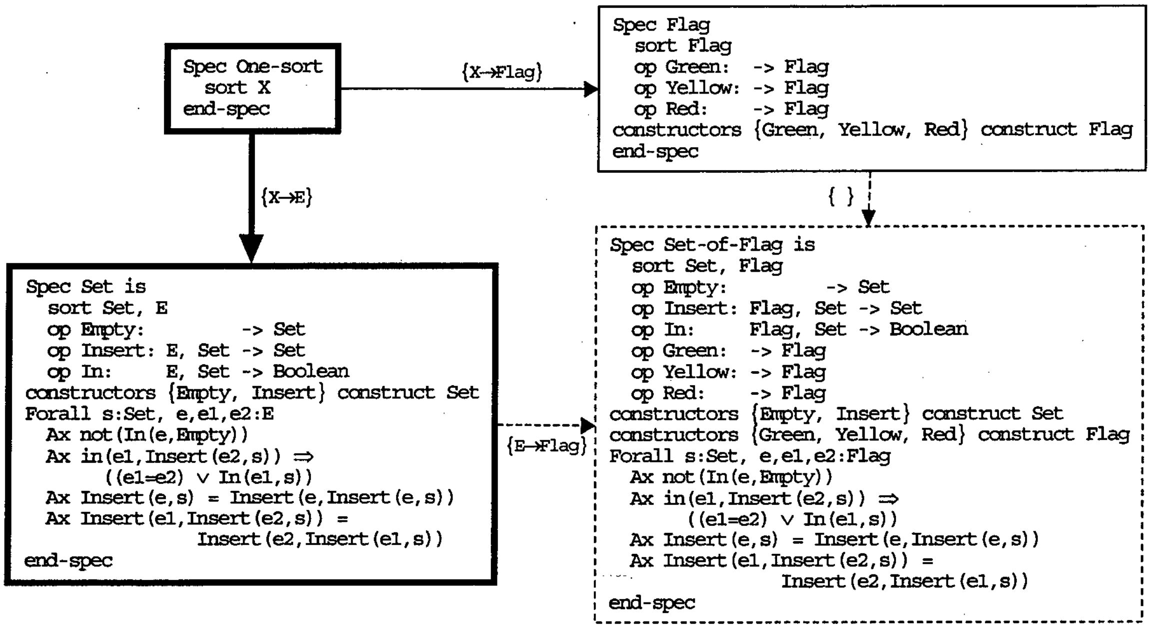
In this section the various forms of category-theory-based parameterization are described in terms of parameterized diagrams (Section 4.2).

E.1.1 Parameterization by a single parameter

The most basic form of category-theory-based parameterization uses only a single parameter. In single-parameterization, a parameterized specification is a morphism [BG77, EL78, Gan83, and EM85] between a formal parameter specification and a body specification, $\mathrm{spec}_{\mathrm{Formal}} \rightarrow \mathrm{spec}_{\mathrm{Body}}$. The parameterized specification is instantiated by developing a morphism from the formal parameter specification to an actual parameter specification, $\mathrm{spec}_{\mathrm{Formal}} \rightarrow \mathrm{spec}_{\mathrm{Actual}}$. This morphism accomplishes two things. It ensures that the actual parameter specification has at least as much structure and properties as the formal parameter specification. Second, together with the morphism to the body specification, it forms the diagram $\mathrm{spec}_{\mathrm{Body}} \leftarrow \mathrm{spec}_{\mathrm{Formal}} \rightarrow \mathrm{spec}_{\mathrm{Actual}}$. The colimit object of that *Spec* diagram is a specification.

Single-parameterization is used in the specification languages Clear [BG77, BG80] and ACT ONE [EM85, Cla89]. Different specification languages and researchers have placed (or proposed) different constraints on parameterized specifications for a variety of reasons. The most common constraint is that the morphism between the formal parameter and the body be a conservative extension (also called a persistent enrichment or theory embedding in the literature). The reasons for this are discussed in Section C.1.4. Because this type of morphism is conserved under pushout, the morphism between actual parameter and instantiated body will also be a conservative extension.

An example of a single-parameterization and instantiation is depicted in Figure E-1 using the Specware language [SJ95, SLM98]. The bolded portion of the diagram is the "parameterized specification". Specification One-Sort is the formal parameter and specification Set is the parameterized body. The morphism from specification One-Sort to specification Set links the formal parameter specification to the body specification and ensures that the formal parameter theory is embedded in the body theory. The specification Flag is the actual parameter to the parameterized specification. The morphism from the formal parameter specification One-Sort to the actual parameter specification Flag ensures that the actual parameter meets the requirements of the formal parameter and represents "parameter passing". The pushout object of the resulting diagram, specification Set-of-Flag, is the instantiated parameterized specification. Using a different actual parameter and performing a pushout results in a different instantiated Set specification. Figure E-2 depicts a "parameterized specification" that has not been instantiated.



```
Spec One-sort
  sort X
end-spec
```

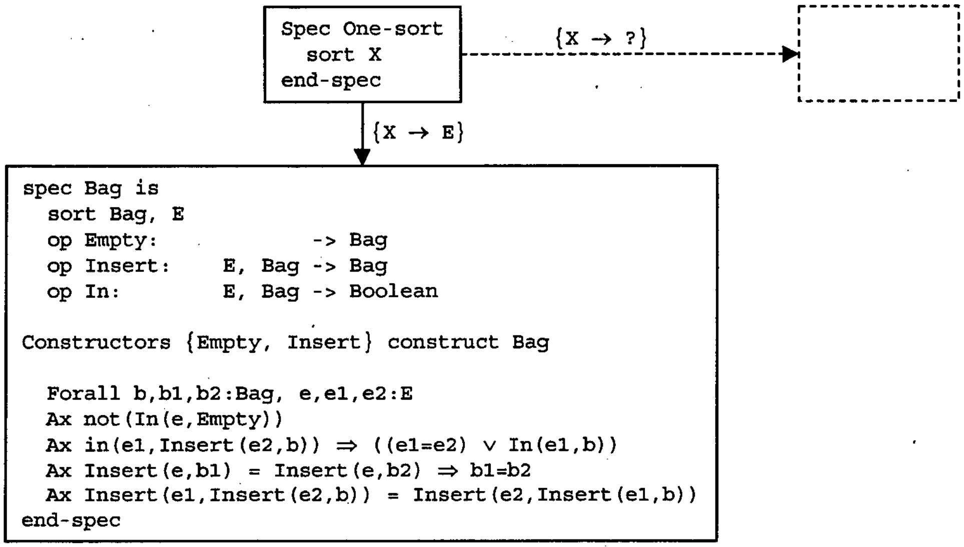
```
Spec Flag
  sort Flag
  op Green:  -> Flag
  op Yellow: -> Flag
  op Red:    -> Flag
constructors {Green, Yellow, Red} construct Flag
end-spec
```

```
Spec Set is
  sort Set, E
  op Empty:         -> Set
  op Insert: E, Set -> Set
  op In:     E, Set -> Boolean
constructors {Empty, Insert} construct Set
Forall s:Set, e,e1,e2:E
  Ax not(In(e,Empty))
  Ax in(e1,Insert(e2,s)) ⇒
       ((e1=e2) ∨ In(e1,s))
  Ax Insert(e,s) = Insert(e,Insert(e,s))
  Ax Insert(e1,Insert(e2,s)) =
              Insert(e2,Insert(e1,s))
end-spec
```

```
Spec Set-of-Flag is
  sort Set, Flag
  op Empty:            -> Set
  op Insert: Flag, Set -> Set
  op In:     Flag, Set -> Boolean
  op Green:  -> Flag
  op Yellow: -> Flag
  op Red:    -> Flag
constructors {Empty, Insert} construct Set
constructors {Green, Yellow, Red} construct Flag
Forall s:Set, e,e1,e2:Flag
  Ax not(In(e,Empty))
  Ax in(e1,Insert(e2,s)) ⇒
       ((e1=e2) ∨ In(e1,s))
  Ax Insert(e,s) = Insert(e,Insert(e,s))
  Ax Insert(e1,Insert(e2,s)) =
              Insert(e2,Insert(e1,s))
end-spec
```

Figure E-1. Instantiation by colimit

Using a morphism between a formal parameter specification and a body specification is too limiting a mechanism to capture the richness needed to reflect the types of parameterization needed in a formal software-engineering environment. The notion of a single specification as a formal parameter forces all "variable" parts of a parameterization to be described within a single formal parameter specification and thus forces all actual parameters to be represented by a single specification. Even simple abstract data types have "multiple" parameters. (An Array has an index type and an element type, and a map has a domain type and a codomain type.) Maps and Arrays and similar abstract data types could only be specified using the limiting single-parameterization mechanism by joining the (possibly) completely independent type parameters into a single formal (and actual) parameter specification.



```
Spec One-sort
  sort X
end-spec
```

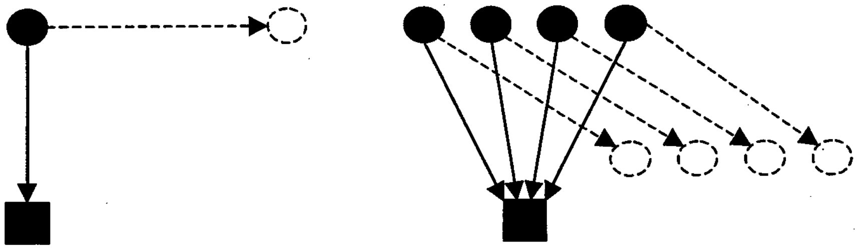
```
spec Bag is
  sort Bag, E
  op Empty:            -> Bag
  op Insert:     E, Bag -> Bag
  op In:         E, Bag -> Boolean

Constructors {Empty, Insert} construct Bag

  Forall b,b1,b2:Bag, e,e1,e2:E
  Ax not(In(e,Empty))
  Ax in(e1,Insert(e2,b)) ⇒ ((e1=e2) ∨ In(e1,b))
  Ax Insert(e,b1) = Insert(e,b2) ⇒ b1=b2
  Ax Insert(e1,Insert(e2,b)) = Insert(e2,Insert(e1,b))
end-spec
```

Figure E-2. Example of single-parameterization

E.1.2 Parameterization by multiple parameters

In [Hax89], a multi-parameterized specification is defined to be a single body specification with multiple parameter specifications where the parameters have no relationship with each other. More formally, a parameterization of multiplicity *n* consists of *n* parameter specifications, P_j, j = 1..n, a body specification, S, and *n* specification morphisms, σ_j:$P_j \rightarrow S$. Instantiation involves creating *n* morphisms to *n* actual parameter specifications, ρ_j: $P_j \rightarrow A_j$, j = 1..n, and taking the colimit of the resulting diagram. The research presented in [Hax89] did not define a convenient syntax for creating or instantiating multi-parameterized specifications. A parameterized specification was formally represented as a collection of morphisms that have a common codomain (a cocone). Each cocone morphism from one of the parameter specifications to the body specification must be a conservative extension morphism

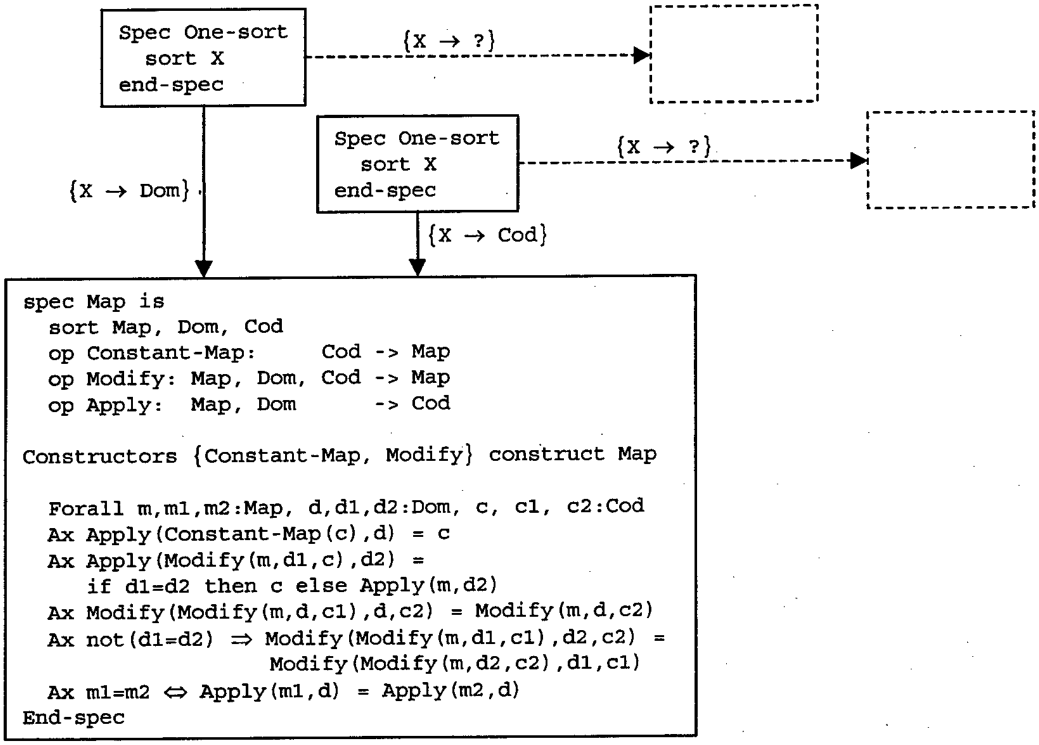
The capabilities of the single-parameterization and multi-parameterization theories are graphically depicted in Figure E-3 using the parameterized diagram notation developed in Section 4.2.3. While single-parameterization enables parameterized objects to be of the shape of Figure E-3(a), multi-parameterization allows parameterized objects to be of the shape of Figure E-3(b). Single-parameterization is just a special case of multi-parameterization. An example of multi-parameterization is depicted in Figure E-4 using the Specware language.



(a) Parameterization by pushout

(b) Parameterization by multiple formal parameters

Figure E-3. Single-parameterization compared with multi-parameterization



```
Spec One-sort
  sort X
end-spec
```

```
Spec One-sort
  sort X
end-spec
```

```
spec Map is
  sort Map, Dom, Cod
  op Constant-Map:      Cod -> Map
  op Modify: Map, Dom, Cod -> Map
  op Apply:  Map, Dom      -> Cod

Constructors {Constant-Map, Modify} construct Map

  Forall m,m1,m2:Map, d,d1,d2:Dom, c, c1, c2:Cod
  Ax Apply(Constant-Map(c),d) = c
  Ax Apply(Modify(m,d1,c),d2) =
     if d1=d2 then c else Apply(m,d2)
  Ax Modify(Modify(m,d,c1),d,c2) = Modify(m,d,c2)
  Ax not(d1=d2) ⇒ Modify(Modify(m,d1,c1),d2,c2) =
                   Modify(Modify(m,d2,c2),d1,c1)
  Ax m1=m2 ⇔ Apply(m1,d) = Apply(m2,d)
End-spec
```

Figure E-4. Example of multi-parameterization

The OBJ3 specification language [GWM92] has a sophisticated parameterization mechanism that can create and instantiate multi-parameterized specifications. As an example, the OBJ code in Figure E-5 consists of (using OBJ terminology) a theory ONE-SORT, an object BITS, a view OS-BITS of how the object BITS satisfies the theory ONE-SORT, a parameterized object, and the instantiation PAIR-of-BITS. In the example, two copies of the "specification" ONE-SORT parameterize "specification" PAIR.

```
th ONE-SORT is
  sort X .
endth
```

```
obj BITS is
  sorts Bit Bits .
  subsorts Bit < Bits .
  ops 0 1 : -> Bit .
  op _ _ : Bit Bits -> Bits .
endo
```

```
view OS-BITS
        from ONE-SORT to BITS is
  sort X to Bits .
endv
```

```
obj PAIR[P1 P2:: ONE-SORT] is
  sort pair
  op <<_;_>> : X.P1 X.P2 -> pair.
  op 1*_ : pair -> X.P1
  op 2*_ : pair -> X.P2
  var e1 : X.P1 .
  var e2 : X.P2 .
  eq 1* <<e1 ; e2 >> = e1 .
  eq 2* <<e1 ; e2 >> = e2 .
endo
```

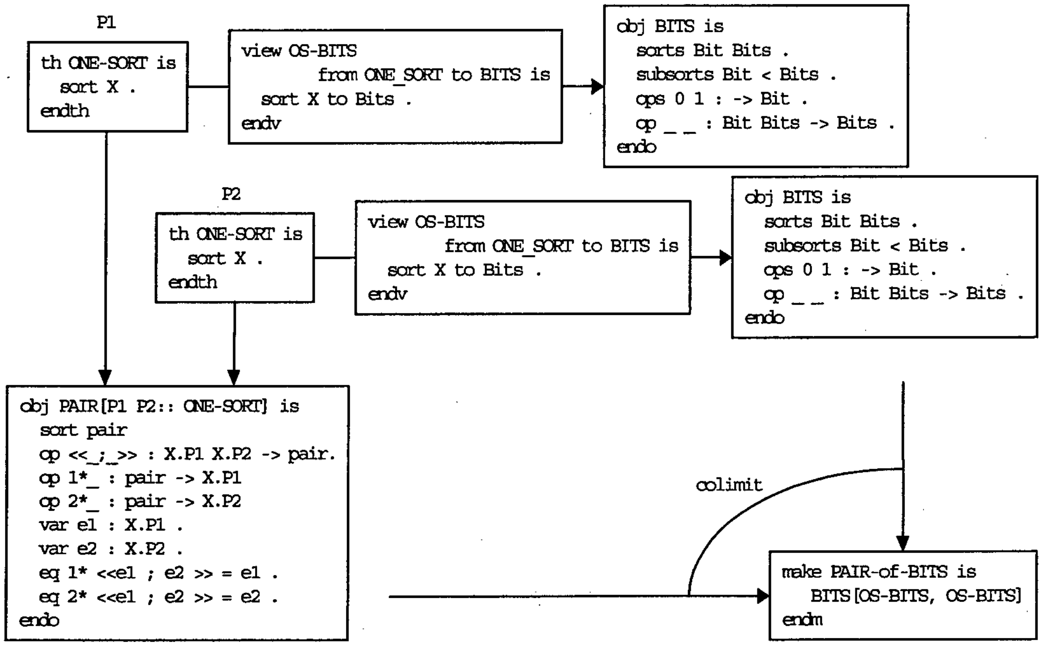
```
make PAIR-of-BITS is
   BITS[OS-BITS, OS-BITS]
endm
```

Figure E-5. OBJ multi-parameterization example

Although the code in Figure E-5 appears to be a syntactic sugar version of λ-style parameterization, it is actually a (syntactic sugar) version of category-theory-based parameterization as the underlying parameterization mechanism involves the use of category theory and not functions and function composition.

The OBJ code in is depicted as a diagram of specifications and morphisms in Figure E-6 in order to demonstrate the underlying category theory semantics. The OBJ theory ONE-SORT and object BITS are essentially specifications with different underlying model theories. The view OS-BITS is essentially the morphism ONE-SORT→BITS. The parameterized object PAIR can be thought of as both a specification and as a diagram of specifications and morphisms. The diagram PAIR is ONE-SORT→PAIR←ONE-SORT where both ONE-SORT diagram nodes have inclusion morphisms to the specification PAIR. Finally, specification PAIR-of-BITS is the colimit of the diagram BITS←ONE-SORT→PAIR←ONE-SORT→BITS that was formed by "instantiating" the PAIR diagram (twice) with the OS-BITS morphism.

OBJ has executable semantics and is not used for refinement. Because of the execution semantics there is no need for multiple copies of a specification in the resulting colimit specification. Thus, for example, the two BITS specifications in Figure E-6 would be combined so that there is only one copy in the resulting colimit specification.



```
th ONE-SORT is
  sort X .
endth
```

```
view OS-BITS
        from ONE_SORT to BITS is
  sort X to Bits .
endv
```

```
obj BITS is
  sorts Bit Bits .
  subsorts Bit < Bits .
  ops 0 1 : -> Bit .
  op _ _ : Bit Bits -> Bits .
endo
```

```
th ONE-SORT is
  sort X .
endth
```

```
view OS-BITS
        from ONE_SORT to BITS is
  sort X to Bits .
endv
```

```
obj BITS is
  sorts Bit Bits .
  subsorts Bit < Bits .
  ops 0 1 : -> Bit .
  op _ _ : Bit Bits -> Bits .
endo
```

```
obj PAIR[P1 P2:: ONE-SORT] is
  sort pair
  op <<_;_>> : X.P1 X.P2 -> pair.
  op 1*_ : pair -> X.P1
  op 2*_ : pair -> X.P2
  var e1 : X.P1 .
  var e2 : X.P2 .
  eq 1* <<e1 ; e2 >> = e1 .
  eq 2* <<e1 ; e2 >> = e2 .
endo
```

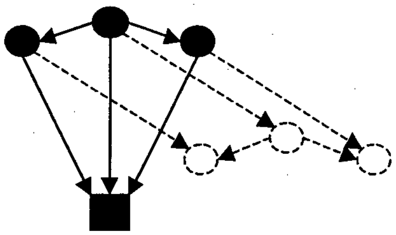
```
make PAIR-of-BITS is
    BITS[OS-BITS, OS-BITS]
endm
```

Figure E-6. Category theory semantics of OBJ multi-parameterization

E.1.3 Parameterization by a diagram of parameters

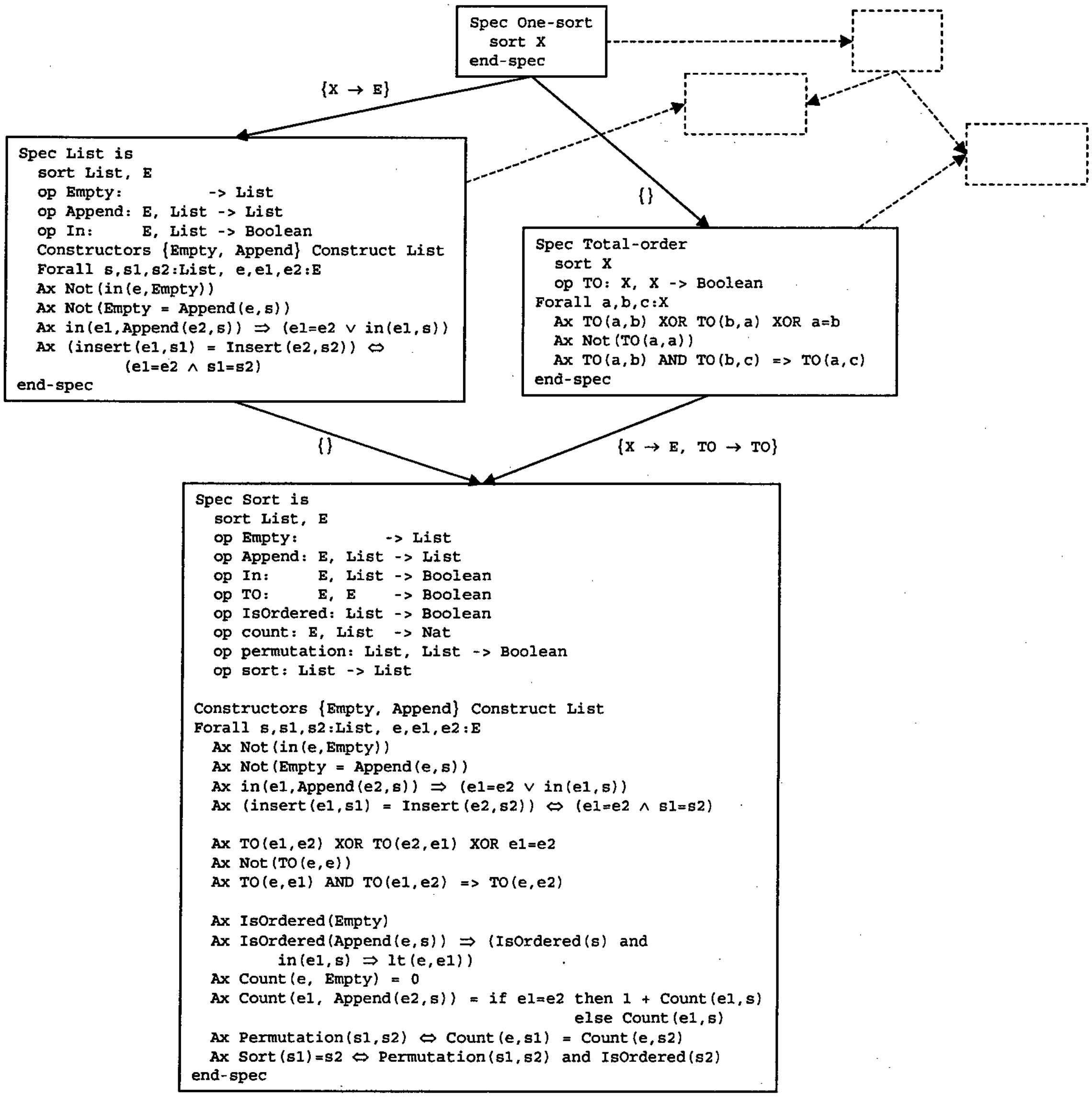
The formal parameter specifications in multi-parameterization are completely independent of each other. Parameterization by a diagram of parameters enables the formal and actual parameters to be structured, i.e. related to each other. In [Dim98], parameterization consists of a body S, a diagram P, and a cocone P→S. Parameterization by a diagram of formal parameters can be viewed as a generalization of multi-parameterization where the formal parameters (and actual parameters) are related to each other. As with multi parameterization, there must be a conservative extension morphism from each specification in the formal parameter diagram to the body specification. There must also be a conservative extension morphism from the colimit of the parameterizing diagram P to the body specification.

In order to instantiate the parameterization by a diagram of parameters, there must exist a family of compatible morphisms to the instantiating specifications as depicted in Figure E-7. There does not exist a high-level specification language syntax for creating and instantiating specifications that are parameterized by diagrams as depicted in Figure E-7. An example of parameterization by a diagram of parameters is depicted in Figure E-8 using the Specware language.



Parameterization by a diagram
(multiple related formal parameters)
and a cocone of morphisms

Figure E-7. Parameterization by a diagram of parameters



```
Spec One-sort
  sort X
end-spec
```

```
Spec List is
  sort List, E
  op Empty:          -> List
  op Append: E, List -> List
  op In:     E, List -> Boolean
  Constructors {Empty, Append} Construct List
  Forall s,s1,s2:List, e,e1,e2:E
  Ax Not(in(e,Empty))
  Ax Not(Empty = Append(e,s))
  Ax in(e1,Append(e2,s)) ⇒ (e1=e2 ∨ in(e1,s))
  Ax (insert(e1,s1) = Insert(e2,s2)) ⇔
           (e1=e2 ∧ s1=s2)
end-spec
```

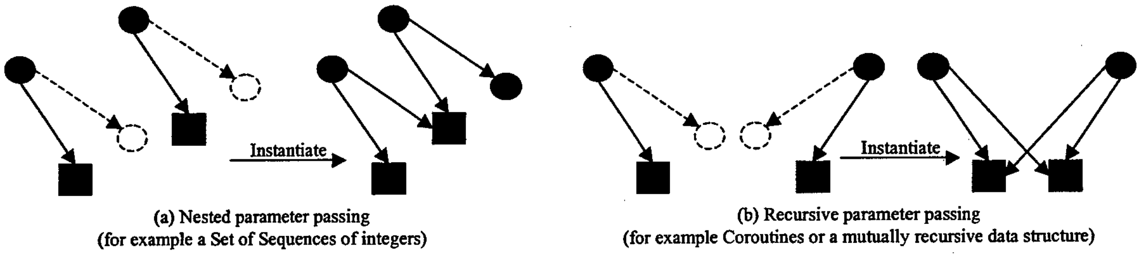
```
Spec Total-order
  sort X
  op TO: X, X -> Boolean
Forall a,b,c:X
  Ax TO(a,b) XOR TO(b,a) XOR a=b
  Ax Not(TO(a,a))
  Ax TO(a,b) AND TO(b,c) => TO(a,c)
end-spec
```

```
Spec Sort is
  sort List, E
  op Empty:          -> List
  op Append: E, List -> List
  op In:     E, List -> Boolean
  op TO:     E, E    -> Boolean
  op IsOrdered: List -> Boolean
  op count: E, List  -> Nat
  op permutation: List, List -> Boolean
  op sort: List -> List

Constructors {Empty, Append} Construct List
Forall s,s1,s2:List, e,e1,e2:E
  Ax Not(in(e,Empty))
  Ax Not(Empty = Append(e,s))
  Ax in(e1,Append(e2,s)) ⇒ (e1=e2 ∨ in(e1,s))
  Ax (insert(e1,s1) = Insert(e2,s2)) ⇔ (e1=e2 ∧ s1=s2)

  Ax TO(e1,e2) XOR TO(e2,e1) XOR e1=e2
  Ax Not(TO(e,e))
  Ax TO(e,e1) AND TO(e1,e2) => TO(e,e2)

  Ax IsOrdered(Empty)
  Ax IsOrdered(Append(e,s)) ⇒ (IsOrdered(s) and
       in(e1,s) ⇒ lt(e,e1))
  Ax Count(e, Empty) = 0
  Ax Count(e1, Append(e2,s)) = if e1=e2 then 1 + Count(e1,s)
                                         else Count(e1,s)
  Ax Permutation(s1,s2) ⇔ Count(e,s1) = Count(e,s2)
  Ax Sort(s1)=s2 ⇔ Permutation(s1,s2) and IsOrdered(s2)
end-spec
```

Figure E-8. Example of parameterization by a diagram of parameters

E.1.4 Nested parameter passing and recursive parameter passing

Nested parameter passing involves passing a parameterized object as an actual parameter, i.e. the body of one parameter object is passed as the actual parameter to another parameterized object. Recursive parameter passing involves the formal parameters of two (or more) parameterized objects being instantiated to each other's body [Hax89]. Most if not all specification languages can handle nested parameter passing as depicted in Figure E-9 diagram (a), but not recursive parameter passing as depicted in Figure E-9 diagram (b).



(a) Nested parameter passing
(for example a Set of Sequences of integers)

(b) Recursive parameter passing
(for example Coroutines or a mutually recursive data structure)

Figure E-9. Nested parameter passing compared with recursive parameter passing

E.2 Lambda-style parameterization

The syntax "λ `x:spec`$_{\texttt{Param}}$ | `spec`$_{\texttt{Body}}$" identifies x as the formal parameter name, $\text{spec}_{\text{Param}}$ as the formal parameter requirements and $\text{spec}_{\text{Body}}$ as the parameterized specification body. Both the requirement and body specifications are built using specification building operations that may reference the specification-valued variable x. Passing a parameter (actual specification) to the λ-expression binds the variable x to the actual specification parameter. (Essentially establishing a morphism from the actual parameter to the formal parameter.) Thus λ x:$\text{spec}_{\text{Param}}$ | $\text{spec}_{\text{Body}}$ is a function from specifications meeting the formal parameter requirements to specifications produced by the specification building operations of $\text{spec}_{\text{Body}}$. λ-style parameterization is used in the specification languages ASL [SW83, Wir86], COLD [Jon89, FJ92], and Spectrum [BFG93].

An example of λ-style parameterization is depicted in Figure E-10. In this example the actual specification that is passed as a parameter is enriched with additional sorts, operations and axioms. `Enrich` is a function that accepts a specification valued parameter and returns a specification. In general any specification transformation operation could be applied to the actual parameter specifications such as "union" that combines specifications and "derive" that allows operations and sorts to be hidden. This style of parameterization is not discussed further as the language and syntax of λ-style parameterization does not fit with the type of categorical constructs used in this dissertation.

```
VAL = Enrich Bool by
      Sorts val
      Ops eq: val, val -> bool
      Forall x,y:val
      Ax eq(x,x)
      Ax Not(x=y)  ⇒ Not(eq(x,y))

SET = λX: VAL. Enrich X by
               Sorts set
               Ops Empty: -> set
                   Insert: set, val -> set
                   In: set, val -> bool
               Forall s:set, x,y:val
               Ax Insert(Insert(s,x),y) = Insert(Insert(s,y),x)
               Ax Insert(Insert(s,x),x) = Insert(s,x)
               Ax Not(In(empty,x))
               Ax In(x(insert(s,y)) = In(s,x) OR eq(x,y)
```

Figure E-10. λ-style parameterization

While the "body" in λ-style parameterization can be more then a single specification (it can be a collection of operations, the result of instantiation is still a specification. This makes λ-style parameterization unsuitable for constructing large structured specifications.

```
diagram Bag-of-Pairs[A,B] is
  import Pair[], Bag[]
  instantiate
    Pair[A, Bag[B]]
end-diagram

diagram Map-to-Bag is
Nodes P1:One-Sort, P2:One-Sort, A:One-Sort, B:One-Sort, Cod:One-Sort,
      Map, Bag, Pair
  Arcs P1 -> A :   {}
       P2 -> B :   {}
       A -> Pair:  {X -> A},
       A -> Map:   {X -> dom},
       B -> Pair:  {X -> B},
       B -> Bag:   {X -> E},
       Cod -> Map: {X -> cod},
       Cod -> Bag: {X -> bag}
end-diagram

spec BagOfPairs-as-MapToBag is
  import colimit of Map-to-Bag
  op Empty: -> Map
  op Insert: Pair, Map -> Map
  op In:     Pair, Map -> Boolean
Constructors {Empty, Insert} construct Map
Forall a:A, b:B, m:Map
Define Empty by
Ax Empty = Constant-Map(Bag.Empty)
Define Insert by
  Ax Insert(<a,b>, m) = Modify(m,a,Bag.Insert(b,apply(m,a)))
Define In by
  Ax In(<a,b>,m) = Bag.In(b,Apply(m,a))
Define Count by
  Ax Count(<a,b>,m) = Bag.Count(b,Apply(m,a))
end-spec

diagram BagOfPairs-as-MapToBag is
  nodes A:One-Sort, B:One-sort, Body:BagOfPairs-as-MapToBag
  arcs A -> Body: {X-> map.dom}
      B -> Body: {X -> Bag.E}
End-diagram

dMorphism BP-to-BPasMB: BagofPairs -> BagOfPairs-as-MapToBag is
  P1 -> A: {}
  P2 -> B: {}
  Bag.One-Sort -> bagOfPairs-as-MapToBag: {X->Pair}
  Pair -> bagOfPairs-as-MapToBag: {}
  Bag -> bagOfPairs-as-MapToBag: {Bag -> Map}

dInterpretation BagOfPair-to-MapToBag: BagOfpairs => MapToBag is
mediator BagOfPairs-as-MapToBag
dom-to-med BP-to-BPasMB
cod-to-med A -> A: {}
           B -> B: {}
           Pair -> BagOfPairs-as-MapToBag: colimit-extension-morphism
           Etc.
```

BagOfPairs-as-MapToBag interpretation code

```
Spec Total-order is
  sort X
  op TO: X, X -> Boolean
Forall a,b,c:X
  Ax TO(a,b) XOR TO(b,a) XOR a=b
  Ax Not(TO(a,a))
  Ax TO(a,b) AND TO(b,c) => TO(a,c)
end-spec

diagram Set-TO is
  import Set[]
  nodes TO
  instantiate
    Set[To.x]
End-diagram

spec triple is
  sort triple, A, B, C
  op make-triple: A,B,C -> triple
  op Project-A: triple -> A
  op Project-B: triple -> B
  op Project-C: triple -> C
  constructors {make-triple}
                   construct triple
  Forall a:A, b:B
  ax project-A(make-triple(a,b,c)) = a
  ax project-B(make-triple(a,b,c)) = b
  ax project-C(make-triple(a,b,c)) = c
end-spec

diagram triple [A,B,C] is
  nodes triple,
        P1:One-Sort, P2:One-Sort, P3:One-Sort
  arcs P1->triple:  {X-> A},
       P2->triple:  {X-> B},
       P3->triple:  {X-> C},
       P1->?:  {X-> A},
       P2->?:  {x-> B}
       P3->?:  {x-> C}
end-diagram

spec Tree-structure is
  import colimit diagram is
                   import triple[], coproduct[]
                   nodes nil, TO
                   instantiate
                     coproduct[nil,triple]
                     triple[cop, TO.X, cop]
                 end-diagram
end-spec
```

Set-TO-as-Tree I

```
spec Tree is
sort Tree,E
  sort-axiom Tree=cop
  sort-axiom E = X
  op Empty: -> Tree
  op Add:   E, Tree -> Tree
  op In:    E, Tree -> Boolean
  op Is-Empty: Tree -> Boolean
constructors {Empty, Insert} construct Tree

Forall t:Tree, e:E
define Empty by
  ax Empty = embed-A(nil)
define Is-Empty by
  ax Is-Empty(t) = (t = empty)
define Add by
  ax Add(e,t) = if Is-Empty(t)
                  then make-triple(nil,e,nil)
                  else if TO(e,Project-B(t))
     then make-triple(Add(e,Project-A(t)), project-B(t), project-C(t))
     else make-triple(Project-A(t), project-B(t), Add(e,project-C(t)))
define In by
  ax In(e,t) = If Is-Empty(t)
                 then FALSE
                 else if e = Project-B(t)
                        then TRUE
                        else if TO(e,Project-B(t))
                               then In(e,Project-A(t))
                               else In(e,Project-C(t))
end-spec

spec SetAsTree is
  import Tree
  sort Set
  sort-axiom Set = Tree
  op Insert
Forall s:Set, e:E
define insert by
  ax Insert(e,t) = if in(e,t) then t
                   else Add(e,t)
end-spec

diagram Tree[E, TO] is
  nodes Tree, TO
  arcs TO->Tree: {X->E}
       TO->?: {X->E, TO->TO}
end-diagram

diagram SetAsTree is
  nodes SetAsTree, P:TO, I:TO
  arcs P->SetAsTree: {X->E}
       P->I: { }
end-diagram
```

Set-TO-as-Tree II

```
dInterpretation Set-TO-to-Tree:Set-TO -> Treed
  mediator SetAsTree
  dom-to-med One-Sort->P.TO: {},
             TO->I.TO: {},
             Set-> SetAsTree: {}
  cod-to-med TO->TO:{},
             Tree->Tree: import-morphism
```

Set-TO-as-Tree III

Bibliography

[AHS90] Jirí Adámek, Horst Herrlich, and George Strecker, *Abstract and Concrete Categories*, John Wiley and Sons, 1990.

[Bac88] R. Back, "A Calculus of Refinements for Program Derivations," *Acta Informatica*, 25:593-624, 1988.

[BCG83] R. Balzer, T. Cheatham, and Green, C., "Software Technology in the 1990's: Using a New Paradigm," *IEEE Computer*, 16(11):39-45, November 1983.

[BD77] R. M. Burstall and J. Darlington, "A Transformation System for Developing Recursive Programs," *Journal of the ACM*, 24:44-67, 1977.

[BFG93] M. Broy, C. Facchi, R, Grosu, et. al. *The Requirement and Design Specification Language SPECTRUM: An Informal Introduction*, Technical Report, TUM-19311 Institut für Informatik, Technische Universität München, 1993.

[BG77] Rod Burstall and Joseph Gougen, "Putting theories together to make specifications," *Proceedings of the 5th International Conference on Artificial Intelligence*, pages 1045-1058, 1977.

[BG80] Rod Burstall and Joseph Gougen, "The Semantics of CLEAR, a Specification Language," Lecture Notes in Computer Science 86, pages 292-332, Springer, 1980.

[BG91] Lee Blaine and Allen Goldberg, "DTRE- A Semi-Automatic Transformation System," *Constructing Programs from Specifications,* B. Möller ed. North-Holland, 1991.

[BGG94] Lee Blaine, Li-Mei Gilham, Allen Goldberg, Richard Jüllig, Jim McDonald, Y. V. Srinivas, *SLANG Language Manual Specware^{TM} Version Core4*, Kestrel, October 29, 1994.

[BKL91] M. Bidoit, H,-J. Kreowski, P. Lescanne, F. Orejas, D. Sannella (eds), *Algebraic System Specification and Development: A survey and annotated Bibliography*, Lecture Notes in Computer Science 501, Springer-Verlag, 1991.

[BM92] L. M. Barroca and J. A. McDermid, "Formal Methods: Use and Relevance for the development of Safety-Critical Systems," *The Computer Journal*, 35(6):579-599, 1992.

[Cla89] I. Claßen, "Revised ACT ONE: categorical constructions for an algebraic specification language," in *Proceedings of the workshop on Categorical Methods in Computer Science with Aspects from Topology*, Lecture Notes in Computer Science 393,pages 124-141, Springer, 1989.

[CW96] Edmund M. Clarke and Jeannette M. Wing, "Formal Methods: State of the Art and Future Directions," *Association for Computing Machinery Computing Surveys*, 28(4) 1996.

[Dij76] Edward W. Dijkstra, *A Discipline of Programming*, Prentice-Hall, 1976.

[Dim98] Theodosis Dimitrakos, *Formal support for specification design and implementation*, PhD thesis, Imperial College, March 1998.

[Ehr82] H. D. Ehrich, "On the theory of specification, implementation and parameterization of abstract data types," *Journal of the Association for Computing Machinery*, 29:206-207, 1982.

[EM85] Hartmut Ehrig and Bernd Mahr, *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*, Springer, 1985.

[EM90] Hartmut Ehrig and Bernd Mahr, *Fundamentals of Algebraic Specification 2: Module specifications and constraints*, Springer, 1990.

[EMCO92] H. Ehrig, B. Mahr, I. Claßen, and F. Orejas, "Introduction to Algebraic Specification. Part 1: Formal Methods for Software Development," *The Computer Journal*, 35(5):460-467, 1992.

[EMO92] H. Ehrig, B. Mahr, and F. Orejas, "Introduction to Algebraic Specification. Part 2: Form Classical View to Foundations of System Specifications," *The Computer Journal*, 35(5):468-477, 1992.

[Fea82] Martin Feather, "A System for Assisting Program Transformation," ACM *Transactions on Programming Languages and Systems,* 4:1-20, 1982.

[FJ90] John S. Fitzergald and Cliff B. Jones, *Modularizing the formal description of a database system*, Technical report UMCS-90-1-1, Dept. of Computer Science, University of Manchester, 1990.

[FJ92] L. Feijs and H. Jonkers, *Formal Specification and Design*, Cambridge Tracts in Theoretical Computer Science 35, Cambridge University Press, 1992.

[Gan83] Harald Ganzinger, "Parameterized Specifications: Parameter Passing and Implementation with Respect to Observability," *ACM Transactions on Programming Languages and Systems*, 5(3):318-354, July 1983.

[Gau93] Marie-Claude Gaudel, "Algebraic specifications," chapter 22 of the *Software Engineer's Reference Book*, ed. John McDermid, CRC Press Inc, 1993.

[Gol84] Robert Goldblatt, *Topoi: The Categorical Analysis of Logic*, North-Holland, Amsterdam, 1984.

[Gra96] Robert P. Graham Jr., *Algebraic Algorithm Design and Local Search*, PhD Dissertation, Air Force Institute of Technology, 1996.

[GB84] Joseph Gougen, and Rod Burstall, "Some Fundamental Algebraic Tools for the Semantics of Computation," *Theoretical Computer Science*, 31, 1984.

[GH93] John V. Guttag, James J. Horning, *Larch: Languages and Tools for Formal Specification*, Springer-Verlag, 1993.

[GM93] M. J. C. Gordon, T. M. Melham, *Introduction to HOL: a theorem proving environment for higher order logics*, Cambridge University Press, 1993.

[GW88] Joseph Gougen, and Timothy Winkler, *Introducing OBJ3*, SRI International Technical Report, SRI-CSL-92-03, March 1992.

[GWM92] Joseph Gougen, Timothy Winkler, Josė Meseguer, et. al., *Introducing OBJ*, Technical Report SRI-CSL-92-03, SRI International, 1992.

[Hax89] Anne Elisabeth Haxhausen, "Parameterized Algebraic Domain Equations," in Lecture Notes in Computer Science 393 *Categorical Methods in Computer Science with Aspects from Topology*, Springer-Verlag 1989.

[Hay92] I. J. Hayes, "VDM and Z: A Comparative Case Study," *Formal Aspects of Computing*, 4(1):76-99, 1992.

[HB94] Thomas Hartrum and Paul Bailor, *A Formal Extension to Object Oriented Analysis Using Z*, Air Force Institute of Technology Technical report, AFIT/EN/TR-94-07, October 1994.

[Jon89] H. Jonkers, "An Introduction to COLD-K," M. Wirsing, J. A. Bergstra eds., *Algebraic Methods: Theory, Tools and Applications*, Lecture Notes in Computer Science 394, pages 139-205, Springer-Verlag, 1989.

[Jon90] C. B. Jones, *Systematic Software Development using VDM.* Prentice-Hall, Second Edition, 1990.

[JOE94] Rosa M. Jiminez, Fernando Orejas, and Hartmut Ehrig, "Compositionality and compatibility of parameterization and parameter passing in specification languages," Mathematical Structures in Computer Science, 5:283-314, 1995.

[JS93] Richard Jüllig and Yellamraju V. Srinivas, "Diagrams for Software Synthesis," *Proceedings of the 8th Knowledge-Based Software Engineering Conference (KBSE)*, Chicago, IL. IEEE Computer Society Press, 1993.

[Klo92] J. W. Klop, "Term Rewriting Systems," *Handbook of Logic in Computer Science*, volume 2, Oxford Science Publishing, 1992.

[Lin93] Huimin Lin, "Procedural Implementation of Algebraic Specification," *ACM Transactions on Programming Languages*, 15(5):876-895, Nov 1993.

[Loe87] Jacques Loeckx, "Algorithmic Specifications: A constructive specification Method for Abstract Data Types," *ACM Transactions on Programming Languages*, 9(4):646-685, Oct 1987.

[Man74] Zohar Manna, *Mathematical Theory of Computation*, McGraw-Hill Inc., 1974.

[Mar82] P. Martin-Lïf "Constructive Mathematics and Computer Programming," *Proceedings 6th International Congress for Logic, Methodology, and Philosophy of Science*, 153-175, North-Holland, 1982.

[Mes89] Jose Meseguer, "General Logics," *Logic Colloquium '87*, North-Holland, 275-329,1989.

[Mor90] C. Morgan, *Programming from Specifications.* Prentice-Hall, 1990.

[MB93] S. MacLane and G. Birkhoff, *Algebra*, 3rd edition, Chelsea Publishing Co., New York, NY.,1993.

[MG84] Jose Meseguer and Joseph Gougen, "Initiality, induction, and computability," *Algebraic Methods in Semantics*, Cambridge University Press, 459-541,1984.

[MW92] Zohar Manna and Richard Waldinger, "Fundamentals of Deductive Program Synthesis," *IEEE Transactions on Software Engineering*, 18(8):674-704, August 1992.

[NHWG89] M. Nielsen, K. Havelund, K. R. Wakner, and C. George, "The RAISE Language, Method and Tools," *Formal Aspects of Computing*, 1:85-115, 1989.

[Par90] Helmut A. Partsch, *Specification and Transformation of Programs: A Formal Approach to Software Development*, Springer-Verlag, New York, 1990.

[Pet77] J. Peterson, "Petri Nets," *Computing Surveys*, 9(3), September 1977.

[PS83] H. Partsch and R. Steinbrüggen, "Program Transformation Systems," *Computing Surveys* 15:199-236, 1983.

[San88] Donald Sanella, *A Survey of Formal Software Development Methods.* Technical Report ECS-LFCS-88-56, University of Edinburgh, July 1988.

[Smi90a] Douglas R. Smith, "KIDS: A Semi-Automatic Program Development System," *IEEE Transactions on Software Engineering*, 16(9), September 1990.

[Smi90b] Douglas R. Smith, "KIDS - A Knowledge-Based Software Development System," *Automated Software Design,* M. Lowry and R. McCartney, eds., 483-514, AAAI/MIT Press, 1991.

[Smi93] Douglas R. Smith, "Constructing Specification Morphisms," *Symbolic Computation*, 15(5-6):571-606, May-June 1993.

[Smi97] Douglas R. Smith, "Planware," Tutorial Notes, *Automated Software Engineering Conference (ASE'97)*, Incline Village Nevada, November 3-5, 1997.

[Spi89] J. M. Spivey, *The Z Notation, A Reference Manual*, Prentice-Hall, 1989.

[Sri90] Yellamraju V. Srinivas, *Algebraic Specification: Syntax, Semantics, Structure*, University of California, Irvine, Dept of Computer Science Technical Report 90-15, June 1990.

[Sri97] Yellamraju V. Srinivas, "Refinement of Parameterized Algebraic Specifications," *Proceedings of the IFIP TC2 Working Conference on Algebraic Languages and Calculi*, Le Bischenberg France 1997.

[SA89] D. M. Steier and A. P. Anderson, *Algorithm Synthesis: A Comparative Study,* Springer-Verlag, New York, 1989.

[SB82] William R. Swartout and Robert Balzer, "On the Inevitable Intertwining of Specification and Implementation," *Communications of the ACM*, 25(7):438-440, 1982.

[SJ95] Yellamraju V. Srinivas and Richard Jüllig, "Specware: Formal Support for Composing Software," *Conference on Mathematics of Program Construction*, Kloster Irsee, Germany, July 1995.

[SLM98] *Specware Language Manual*, Version 2.03, Kestrel Development Corp., March 1998.

[SST92] Donald Sanella, Stefan Sokolowski, and Andrzej Tarlecki, "Toward formal development of programs from algebraic specifications: parameterization revisited," *Acta Informatica* 29(8):689-736, 1992.

[ST88] Donald Sanella, and Andrzej Tarlecki, "Toward Formal Development of Programs from Algebraic Specifications: Implementations Revisited," *Acta Informatica*, 25:233-281,1988.

[SUG98] *Specware User Guide*, Version 2.03, Kestrel Development Corp., March 1998.

[SW83] D. Sannella and M. Wirsing, "A Kernel Language for Algebraic Specification and Implementation." In M. Karpinski, ed., *Proceedings of the 11th Colloquium on Foundations of Computer Theory*, Lecture Notes in Computer Science 158 , pages 413-427, Springer-Verlag, 1983.

[Wal97] Richard Waldinger, "Deductive Program Synthesis," Tutorial Notes, *Automated Software Engineering Conference (ASE'97)*, Incline Village Nevada, November 3-5, 1997.

[Wex81] Richard L. Wexelblat, *History of Programming Languages*, from the *ACM SIGPPLAN History of Programming Languages Conference*, June 1978, Academic Press 1981.

[Wil95] Andrew J. Wiles, "Modular Elliptic Curves and Fermat's Last Theorem," *Annals of Mathematics*, Second Series, 141(3), 1995.

[Win90] Jeannette M. Wing, "A Specifier's Introduction to Formal Methods," *IEEE Computer*, 23(9):8-24, September 1990.

[Wir86] M. Wirsing, "Structured Algebraic Specifications: A Kernel Language," *Theoretical Computer Science*, 42:123-249,1986.

[Wir90] M. Wirsing, Algebraic Specification, *Handbook of Theoretical Computer Science*, Ed. J. van Leeuwen, Elsevier Science Publishers, 1990.

[WSGJ96] Richard Waldinger, Yellamraju V. Srinivas, Allen Goldberg, and Richard Jüllig, *Specware^{tm} Language Manual, version 2.0.1*, August 1996.

[Zav96] Pamela Zave, "Formal Methods are Research, Not Development," *IEEE Computer*, 29(4):26-27, April 1996.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY *(Leave blank)*

2. REPORT DATE
September 1999

3. REPORT TYPE AND DATES COVERED
Doctoral Dissertation

4. TITLE AND SUBTITLE
Formal Representation and Application of Software Design Information

5. FUNDING NUMBERS

6. AUTHOR(S)
Thomas M. Schorsch, Major, USAF

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Air Force Institute of Technology
2950 P Street
WPAFB OH 54533-7765

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/DS/ENG/99-08

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Mr. Roy F. Stratton
Air Force Research Laboratory
AFRL/IFTD
525 Brooks Road
Rome, NY 13441-4505 (303) 315-3004

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Formal methods for developing software use mathematical frameworks to specify, develop and verify software systems, especially safety critical systems where error free software is a necessity. A transformation system is a formal method that refines a requirement specification into an implementation by successively adding design decisions in the form of precisely verified design information. Current algebraic representations of design information (specifications, morphisms, and interpretations) and methods for applying algebraic specification design information (diagram refinement) cannot correctly represent and apply design information involving higher-level design information.

This investigation develops innovative methods for constructing and refining structured algebraic requirement specifications, as opposed to individual specifications. A category of diagrams and diagram morphisms is developed and applied to algebraic specifications and morphisms that enables the structure of requirement specifications and design information to be dealt with explicitly. Diagram interpretations enable structured design information to be correctly represented and applied, including the refinement of parameterized diagrams and restructuring refinements. The developed approach enables one to create a library of correctly represented software design information. Software could then be developed directly from requirements by selecting design choices from a library. Such a transformation system would enable correct-by-construction software to be developed rapidly and easily.

14. SUBJECT TERMS
Formal methods, automated software engineering, knowledge-based software engineering, software synthesis, software refinement, design information, algebraic specification, category theory

15. NUMBER OF PAGES
287

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	**18. SECURITY CLASSIFICATION OF THIS PAGE**	**19. SECURITY CLASSIFICATION OF ABSTRACT**	**20. LIMITATION OF ABSTRACT**
Unclassified	Unclassified	Unclassified	UL